Characterizing the evolution of statically-detectable performance issues of Android apps

Das, Teerath; Di Penta, Massimiliano; Malavolta, Ivano

published in
Empirical Software Engineering
2020

DOI (link to publisher)
10.1007/s10664-019-09798-3

document version
Publisher's PDF, also known as Version of record

Empirical Software Engineering (2020) 25:2748–2808
https://doi.org/10.1007/s10664-019-09798-3

Characterizing the evolution of statically-detectable performance issues of Android apps

Teerath Das[1] · **Massimiliano Di Penta**[2] · **Ivano Malavolta**[3]

Published online: 4 May 2020

Abstract

Mobile apps are playing a major role in our everyday life, and they are tending to become more and more complex and resource demanding. Because of that, performance issues may occur, disrupting the user experience or, even worse, preventing an effective use of the app. Ultimately, such problems can cause bad reviews and influence the app success. Developers deal with performance issues thorough dynamic analysis, i.e., performance testing and profiler tools, albeit static analysis tools can be a valid, relatively inexpensive complement for the early detection of some such issues. This paper empirically investigates how potential performance issues identified by a popular static analysis tool — Android Lint — are actually resolved in 316 open source Android apps among 724 apps we analyzed. More specifically, the study traces the issues detected by Android Lint since their introduction until they resolved, with the aim of studying (i) the overall evolution of performance issues in apps, (ii) the proportion of issues being resolved, as well as (iii) the distribution of their survival time, and (iv) the extent to which issue resolution are documented by developers in commit messages. Results indicate how some issues, especially related to the lack of resource recycle, tend to be more frequent than others. Also, while some issues, primarily of algorithmic nature, tend to be resolved quickly through well-known patterns, others tend to stay in the app longer, or not to be resolved at all. Finally, we found how only 10% of the issue resolution is documented in commit messages.

Keywords Android · Mobile performance issues · Mining software repositories · Empirical study

Communicated by: Ahmed E. Hassan

✉ Ivano Malavolta
i.malavolta@vu.nl

Teerath Das
teerath.das@gssi.it

Massimiliano Di Penta
dipenta@unisannio.it

[1] Gran Sasso Science Institute, L'Aquila, Italy

[2] Department of Engineering, University of Sannio, Benevento, Italy

[3] Department of Computer Science, Vrije Universiteit Amsterdam, Amsterdam, The Netherlands

1 Introduction

Mobile applications are becoming more and more important and pervasive in everyone's lives. The underlying economy is estimated to go over $ 950 billion in 2018.[1] In such a context, the hardware vendors are offering more and more powerful devices, in terms of CPU, memory, Graphical Processing Unit (GPU), and equipped sensors. At the same time, apps are becoming more and more resource-demanding. This is especially the case of video games and multimedia applications, although this might be the case for other app categories as well, e.g., for productivity, sport, or other apps. Moreover, inappropriate usage of API or of third-party resources could contribute to introduced performance issues in apps.

Performance issues can negatively affect the user experience and in some cases impede an effective usage of the apps' features. This can ultimately impact the app user ratings and reviews which, unless properly addressed, can negatively impact to the app's success (Palomba et al. 2018).

Dynamic analysis remains the main approach for identifying performance problems and for identifying bottlenecks in the app architecture, some problems can possibly be early detected through the use of static analysis tools during development. Static analysis tools analyze either source code or byte code using different techniques (some rely on simple pattern matching, others employ more sophisticated techniques including data flow analysis) with the aim of identifying, for example, code style issues, potential bugs, sources of potential security issues, or potential performance problems. Examples of such tools include some general-purpose ones, e.g., *FindBugs*[2] or *PMD*,[3] but also some Android-specific tools. These includes *Paprika* (Hecht et al. 2015b), *aDoctor* and *Android Lint*.[4] Among other kinds of potential problems, such tools are also able to identify potential performance issues. For example, Android Lint detects seven kinds of performance issues,

The goal of this paper is to empirically investigate the extent to which (potential) performance issues reported by static analysis tools are actually resolved by developers. More specifically, the study analyzes the occurrence and resolution of seven kinds of performance issues identified by Android Lint in 316 open source Android apps hosted on GitHub (among 724 apps we analyzed in total). We have chosen Android Lint as static analyzer, because it is integrated in Android Studio and also available as an Eclipse plugin, therefore it likely to be used by many Android developers.

First, we identify and report the occurrence of likely performance issues across the analyzed apps. Then, we trace the detected performance issues across the apps' evolution history, and determine whether they have been detected and, if this is the case, what is the distribution of their survival time in the app. This has the purpose of determining whether there are some kinds of issues that tend to be resolved quickly whereas others are more likely to be ignored by developers. Finally, we analyze the extent to which the resolution of the detected performance issues is also acknowledged by developers and documented in commit messages.

So far, performance issues have been investigated in Web applications (Ahmed et al. 2016), heterogeneous environments (Foo et al. 2015), or large-scale applications (Malik et al. 2013). Also, Zaman et al. conducted a qualitative study of performance bugs (Zaman

[1] ACT. Act state of the app enconomy 2018, 2018.

[2] Findbugs™- find bugs in java programs. http://findbugs.sourceforge.net/.

[3] PMD - an extensible cross-language static code analyzer. https://pmd.github.io/.

[4] Android studio project site - Android Lint. http://tools.android.com/tips/lint.

et al. 2012). To the best of our knowledge, there is only work on the analysis of mobile app performance bugs by Liu et al. (2014), limited to the analysis of 70 real bugs.

The paper is organized as follows. Section 2 provides a brief background about static analysis tools for Android, with particular emphasis on Android Lint and about performance issues it is able to detect. Section 3 describes the study definition and planning. Sections 4, 5, 6, 7, and 8 report the study results. Section 9 provide a summary of the results, and discusses implications for developers and researchers, while threats to validity are discussed in Section 10. After presenting related work (Section 11), Section 12 concludes the paper and outlines directions for future work.

2 Static Analysis Tools for Android Apps

Static analysis tools analyze the source code, bytecode, or other artifacts of software applications without executing them, with the aim of extracting facts about a software system (Nielson et al. 2015). For what concerns Android apps, static analysis is gaining a growing interest in both academia and industry (Li et al. 2017; Joorabchi et al. 2013). A large number of static analysis tools exist for Android, ranging from structural and control-flow analysis to data-flow and state-based analysis, interval analysis (used in optimizing compilers) and so on (Li et al. 2017). Regarding the identification of performance-related issues, there is a number of available static analysis tools, which are used to find various types of performance issues. These tools work at different levels of granularity, i.e., some tools work on the source code of the app, others on bytecode and some tools work on APK level. In the following, we report the most widely used tools to identify performance-related issues of Android apps. More specifically, we describe three general-purpose tools for Java applications, i.e., *FindBugs*, *PerfChecker* and *PMD* and three Android-specific tools, *Paprika*, *aDoctor*, and *Android Lint*.

FindBugs is an open-source static analysis tool conceived to identify bad smells/defects in Java applications in general (it is not Android-specific) through the analysis of Java bytecode. *FindBugs* can be executed via Maven or ANT configurations, within the Netbeans or Eclipse IDE, through its dedicated graphical user interface, or via command line. Similarly to *Android Lint*, *FindBugs* can also detect different categories of issues such as correctness, internationalization, security, and performance issues. Each issue is associated with a high, medium, or low severity.[5]

PerfChecker is used to identify two types of performance anti-patterns, i.e., (i) lengthy operations in the main thread of a program, and (ii) violations of the view holder pattern (Liu et al. 2014). Also *PerfChecker* is based on the Soot framework. *PerfChecker* takes the Java bytecode of Android apps as input and generates a warning when it finds any performance related anti-patterns.

PMD is an open-source code analyzer for Java which executes syntactic checks on the source code. It is able to detect issues like bad programming practices, or inefficient code, that can degrade the performance of a program if accumulated.

Paprika can detect performance-related bugs in Android apps (Hecht et al. 2015b). *Paprika* takes as input the APK file of the app, produces a set of metrics related either to the Android programming model (e.g., number of broadcast receivers) or to standard principles of object orientation (e.g., coupling between objects). Once those metrics are computed, *Paprika* allows the developer to query them and to identify potential antipatterns in the app

[5]Findbugs™- find bugs in java programs. http://findbugs.sourceforge.net/.

```
DrawAllocation
--------------
Summary: Memory allocations within drawing code

You should avoid allocating objects during a drawing or layout operation.
These are called frequently, so a smooth UI can be interrupted by garbage
     collection pauses caused by the object allocations.

The way this is generally handled is to allocate the needed objects up front
     and to reuse them for each drawing operation.

Some methods allocate memory on your behalf (such as Bitmap.create) and these
     should be handled in the same way.
```

Listing 1 DrawAllocation description by Android Lint Tool

(e.g., blob class or heavy broadcast receiver Hecht et al. 2015b). *Paprika* is based on Soot, a widely-adopted static analysis framework for Java and Android bytecode.[6]

aDoctor is a fully-automated tool (Palomba et al. 2017) capable of detecting 15 types of Android-specific code smells. The tool analyzes the abstract syntax tree of the source code and applies a set of identification rules, which are based on the descriptions of smells given in the catalog by Reimann et al. (2014). At the time of writing, 7 out of 15 Android code smells are related to performance issues i.e., Internal Getter and Setter (IGS), Leaking Inner Class (LIC), Member Ignoring Method (MIM), etc.

Android Lint analyzes both Java source code as well as XML resource files. Android Lint can be used either as a stand-alone tool or as an IDE plugin (it is included in Android Studio and available for Eclipse). The tool is capable to detect different types of issues such as security, performance, usability, and accessibility. Listing 1 shows the description of the `DrawAllocation` performance check of *Android Lint*.

In this study, we use Android Lint because (i) it covers a relatively large number of recurrent Android performance issues (see Section 4), (ii) it is among the few static analysis tools supporting a dedicated category for Android performance-related issues,[7] (iii) it can be considered as the de facto static analysis tool for Android apps since it is activated by default in Android Studio (i.e., the official development environment for Android apps), where its execution is part of the Java compilation step from the developer's perspective, and (iv) it can be integrated with relatively low effort into a larger software pipeline, like the one we developed in this study (see Section 3.3).

3 Study Design

This study has been carried out by following the guidelines for designing, conducting, and reporting empirical experiments in software engineering (Wohlin et al. 2012; Shull et al. 2007). In this section we focus on the design of our study, specifically we present its goal and research questions (Section 3.1), context selection (Section 3.2), data extraction (Section 3.3).

[6] Soot - A framework for analyzing and transforming Java and Android applications. https://sable.github.io/soot/.

[7] At the time of writing only *FindBugs* also provides a dedicated category for performance-related issues, but it is not specific to Android apps.

Table 1 Goal of this study

Analyze	the change history of Android mobile applications
for the purpose of	characterizing their evolution
with respect to	statically detectable performance issues
from the viewpoint of	developers and researchers
in the context of	open source Android applications

A complete *replication package* is publicly available[8] for allowing independent replication and verification of our study. The replication package includes the Python and shell scripts for identifying the targeted Android apps, the list of all considered GitHub repositories, the mined Google Play metadata, the raw data extracted from each GitHub repository, the Python and Shell scripts for extracting the raw data, and the R scripts we developed for data exploration and analysis, and for visualizing the obtained results.

3.1 Goal and Research Questions

We formulate the goal of this study by using the Goal-Question-Metric perspectives (Basili et al. 1994). Table 1 shows the result of our goal formulation.

In the following we present and discuss the research questions we translated from the above mentioned overall goal.

> $\boldsymbol{RQ_0}$ – To what extent does Android Lint identify performance issues in the analyzed apps?

As already introduced, in this study we exploit Android Lint for identifying performance-related issues of Android apps. This research question is exploratory in nature and aims at characterizing the number, frequency, and distribution of performance issues identified by Android Lint across the versioning history of all apps. By answering RQ_0, we assess whether the context of our study (i.e., the apps dataset we built – see Section 3.2) and Android Lint provide enough data points for answering the remaining research questions. Moreover, by answering RQ_0 we identify the most recurrent statically-detectable performance issues during the evolution of Android apps, providing empirical evidence to developers and researchers for getting a better understanding of Android-specific performance issues.

> $\boldsymbol{RQ_1}$ – How does the number of statically-detectable performance issues of Android apps vary over time?

The main objective of RQ_1 is to investigate whether the evolution of statically-detectable performance issues across different Android apps exhibits identifiable patterns. The identified patterns can be used by researchers as a foundation for investigating the relationships between apps exhibiting the same or different patterns. Also, the emerging patterns can

[8] https://github.com/S2-group/AndroidPerformanceIssues

guide developers in identifying potentially dangerous patterns in their own apps, e.g., a sudden increase of performance issues without any subsequent decrease.

> $\boldsymbol{RQ_2}$ – Which types of statically-detectable performance issues tend to remain in Android apps across their life time?

By answering RQ_2 we provide insights about how each type of Android performance issues tend to remain in Android apps over time, A long survival time of a performance issue can have two completely different explanations. On the one hand, it can indicate that the issue does not seriously affect the app's performance, and therefore it has been ignored by developers. On the contrary, if it is a harmful issue, remaining in the app for a long time would mean potentially affecting multiple releases.

> $\boldsymbol{RQ_3}$ – What is the lifetime of statically-detectable performance issues of Android apps?

With the term lifetime we mean the interval between the introduction and the resolution of a performance issue along the versioning history of the app. The underlying intuition behind RQ_3 is that different types of statically-detectable performance issues have significantly different life spans. The results of RQ_3 can highlight whether particular kinds of performance issues tend to be resolved quicker than others, either because they are easier to spot, or because they are deemed to be more dangerous. Furthermore, we also characterize whether the lifetimes of different types of performance issues follow know probability distributions. This can help developers in knowing how likely a specific performance issue in their app will be resolved from the code base.

> $\boldsymbol{RQ_4}$ – To what extent the resolution of statically-detectable performance issues of Android apps have been documented by developers?

This research question aims at (i) assessing if developers document in their commit messages the resolution of performance issues and (ii) providing a minimal catalog of representative solutions, one for each type of Android performance issue. The results of RQ_4 provide empirical evidence about whether developers consciously document their activities related to the resolution of performance issues. Researchers can use such evidence as a foundation for further studies on the relationship between documented and not-documented activities related to statically-detectable performance issues of Android apps. Finally, developers can use the catalog of solutions for standing on other developers' shoulders and use it as a reference for solving the issues raised by their instance of *Android Lint*.

The research questions discussed above drive the whole study, ranging from the selection of *Android Lint* as analysis tool, to the activities related to apps selection, data extraction, and analysis.

3.2 Context Selection

This study focuses on *real-world Android apps* for which we can execute the Android lint analysis tool across different *versions* of their source code. More specifically, the context

of this study consists of a set of Android apps that (i) have their versioning history hosted on GitHub and (ii) are distributed in the Google Play store. We chose GitHub as target of source code repositories because (i) it is extremely popular among developers (as of June 2018, it has a community of 24 million developers[9]), (ii) it hosts a huge amount of metadata that can be accessed through its API,[10] and (iii) there is a variety of available tools for mining and processing data and metrics from GitHub repositories (e.g., the `git log` and `git diff` tools). We focus on apps distributed on the Google Play Store because it is the official distribution channel of Android apps.

In the remainder of this section, a detailed overview of the process for building the dataset of Android apps is given. Identifying the required target dataset of Android apps for this research requires applying several filtering steps, which are documented alongside the respective numbers of apps resulting from each filtering step. The dataset building process of this study is similar to the one proposed in (Das et al. 2016) and its 10 steps are shown in Fig. 1 All steps (except step ③) are implemented as independently-executable software components (mainly Python and Unix shell scripts) and their source code is available in the replication package of this study. The initial collection of those apps originates from three different sources, namely: FDroid, GitHub, and Wikipedia. The reason why we chose them is because we wanted to achieve a diverse set of sources, including the most popular host for open source (GitHub), a store of open source Android apps (FDroid), and finally an online-compiled catalog (Wikipedia, which was included for the sake of completeness, because as explained below it contributed with a fairly limited number of apps).

The first source for our dataset is FDroid,[11] a well-known online catalog of free and open-source Android projects (step ①). From this catalog, a search is applied that locates apps that contain: a) a link to the respective GitHub repository, and b) a link to the respective Google Play store page. Mining the FDroid repository resulted in 350 potentially relevant GitHub repositories.[12]

From Github (step ②), a custom search targeting all the repositories containing a link to a Google Play Store app page in their readme files is performed. In order to do not occur into the GitHub limit of 1,000 results per search, we stratify our search queries by date range so that each search results in less than 1,000 results. The whole set of considered dates ranges from the creation of GitHub (i.e., Jan 1, 2001) to the day in which the searches were performed (i.e., Feb 15, 2016). This search resulted in a total of 4,788 potentially relevant GitHub repositories.

The third source for our dataset is a Wikipedia[13] page containing a maintained list of free and open-source Android apps (step ③). We manually screened this list of apps to select the ones that, again, contain a link to the respective GitHub repository and are published on the Google Play Store. This step results in a total of 35 potentially relevant GitHub repositories.

In step ④, each repository coming from three data sources is uniquely identified by its $< repository\ owner\ ,\ repository\ name >$ pair and all duplicates are merged. The execution of this step results in a total of 4,287 unique GitHub repositories.

[9] https://github.com/features

[10] https://developer.github.com/v3/

[11] F-Droid - Free and Open Source Android App Repository. https://f-droid.org/en/.

[12] At the time of writing this paper (June 2018) the search functionality on FDroid appears to be broken or not working. Furthermore, the https://f-droid.org/forums/search/ endpoint that was used in the mining script does not exist anymore.

[13] Wikipedia page on open-source Android apps, 2017.

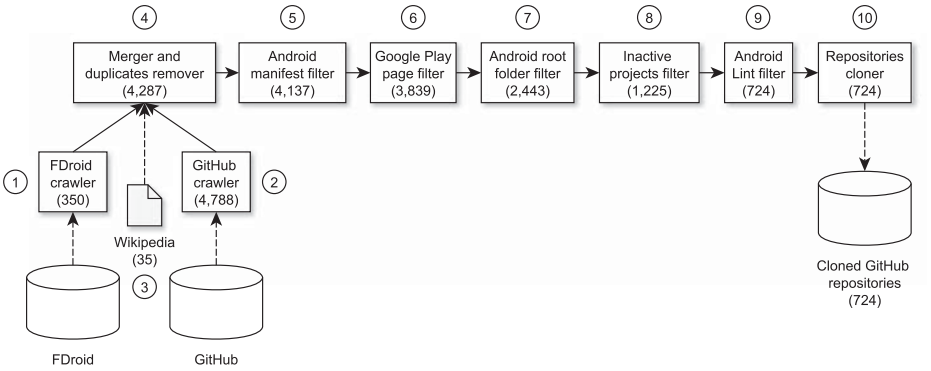


Fig. 1 The dataset creation process

In the next filtering step (step (5)), we identify those repositories which potentially contain the source code of an Android app. This filtering step is done by considering only the repositories containing the mandatory `Android Manifest.xml` file. Indeed, as mentioned on the official Android developers website,[14] an Android app shall always come with an Android manifest file.

The manifest of an Android app contains all the essential information supplied to the Android system, allowing it to run the actual app. For example, the manifest file is in charge of naming the Java package for the app, which serves as an unique identifier of each app. Information contained in the Android manifest file includes the main components of the app, its required permissions, the minimum required API level for the app, the third-party libraries used by the app, etc.

In step (6) we filter out all those repositories containing apps that are not published in the Google Play store. In the context of this process, this may occur if (i) the considered GitHub repository is a simple demo or toy example, which has been developed for personal or internal usage only, (ii) developers removed the app from the Google Play store, or (iii) Google deliberately took down the app because it was violating some of its distribution policies. This step has been done by extracting the package name of the app from its manifest file, and performing a network request to the URL where the Google Play page of the app should be present.[15]

Step (7) involves the identification of the app's root folder containing its source code. The rationale for this step is that the folder containing the Android manifest file should also contain the complete source code for the app. Indeed, in this step we need to exclude those repositories where the manifest file actually refers to an Android library, to the binaries of some other apps, etc. This step is realized by checking if the folder containing the Android manifest file follows the structure mandated by the Android platform.[16] Moreover, this step

[14] https://developer.android.com/guide/topics/manifest/manifest-intro.html

[15] This check is sound since the web page of an app in the Google Play store follows a fixed pattern, i.e., `https://play.google.com/store/apps/details?id=[app package name]`.

[16] Step 5 and Step 7 are redundant, we deliberately decided to keep both of them because during the execution of the dataset creation process we had to experiment with different heuristics in Step 7 and having it as a stand-alone step within the pipeline helped us in easily run it in isolation.

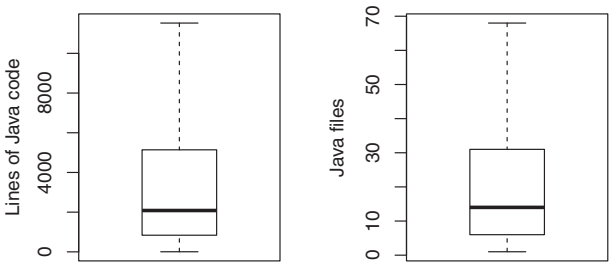


Fig. 2 Size of the apps in the dataset

is crucial for our study as it is possible that some repositories contain code unrelated to the app itself, such as server-side code, design artifacts, etc.; all the steps related to the app source code in the data extraction phase (e.g., running the Android Lint tool — see Section 3.3) are scoped within the app root folder.

Step ⑧ involves the identification and filtering of inactive GitHub repositories. Indeed, it is well known that mining GitHub repositories brings the risk of considering inactive or unmaintained repositories, thus adding noise to the results of the study (Kalliamvakou et al. 2016). For each repository, we extract (i) the *app development lifetime* and (ii) the *number of commits*. The app development lifetime is defined as the range between the first and last commits contributing to the app root folder in the repository, whereas the number of commits is defined as the count of all the commits performed in the app root folder in the repository. In order to avoid inactive or unmaintained repositories (Kalliamvakou et al. 2016), we considered only the apps having a lifetime span of at least 4 weeks and with at least 10 commits.

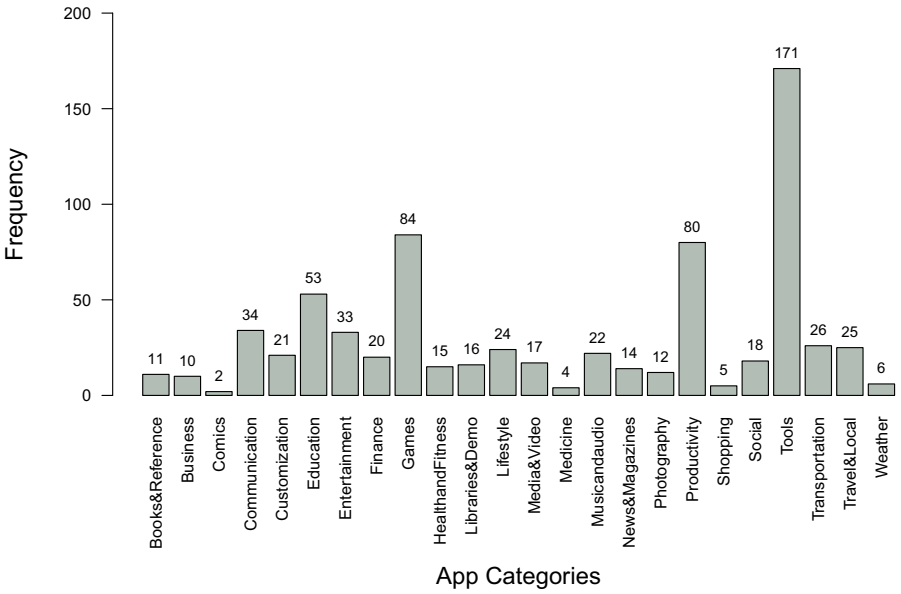


Fig. 3 Google Play categories of the apps in our dataset

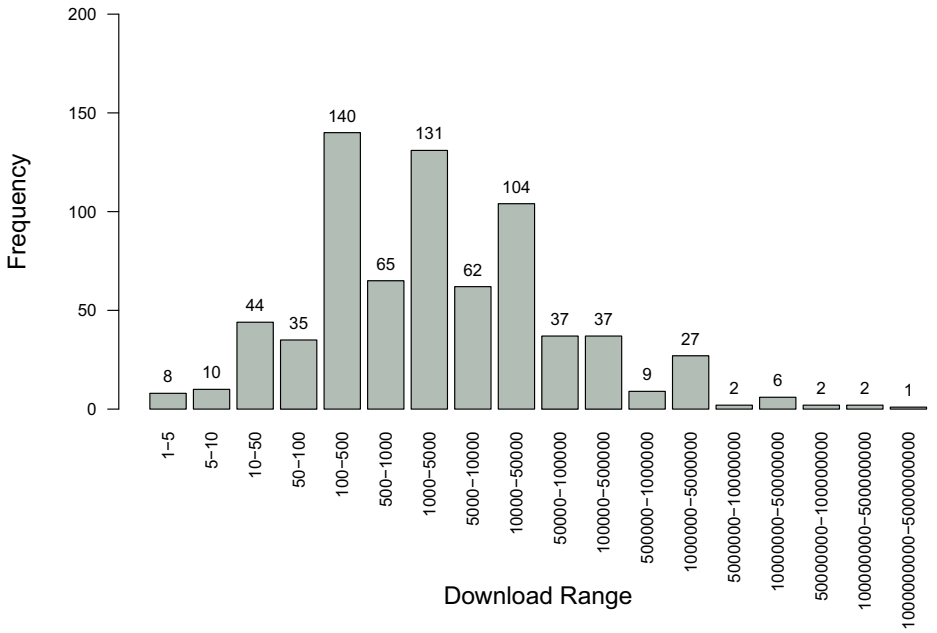


Fig. 4 Google Play download ranges of the apps in our dataset

Step (9) involves the filtering of all those apps which cannot be properly analyzed by the *Android Lint* tool. It is important to note that Android Lint requires that the app under analysis is fully built. Building arbitrary software mined from third-party GitHub repositories is notoriously difficult, mainly due to missing dependencies (e.g., the server hosting a dependency is no longer reachable), Java compilation errors (e.g., undefined symbols, missing packages), and project-specific build commands (e.g., non-default Gradle tasks) (Hassan et al. 2017; Sulír and Porubän 2016). In this step we managed to cover many recurrent non-standard cases by iteratively running the *Android Lint* tool on all repositories and (i) manually refining its configuration and (ii) adding preprocessing steps for making the app and its Gradle configuration more Lint-friendly (see step 2 in Section 3.3). Excluded repositories include apps with very peculiar Gradle configurations, Kotlin-based apps (we focus on Java-specific issues), apps heavily based on the Native Development Toolkit (NDK[17]), unbuildable apps due to missing keystore information.

Finally, in step (10) we locally clone all the selected GitHub repositories. After this process, our final dataset is composed of **724 GitHub repositories containing open, published, and actively maintained Android apps, for which an analyzable commit history is available**. Out of them, a large majority is from Github(630), followed by FDroid (88) and Wikipedia (6).

As shown in Fig. 2, the dataset is quite heterogeneous in terms of both lines of Java code (median = 2083.0, mean = 5325.7, IQR = 4299.75) and number of Java files per app (median = 14, mean = 29.38, IQR = 25.0). Moreover, the dataset also covers 24 different Google Play categories (see Fig. 3 and all downloads ranges (see Fig. 4).

[17] http://developer.android.com/ndk

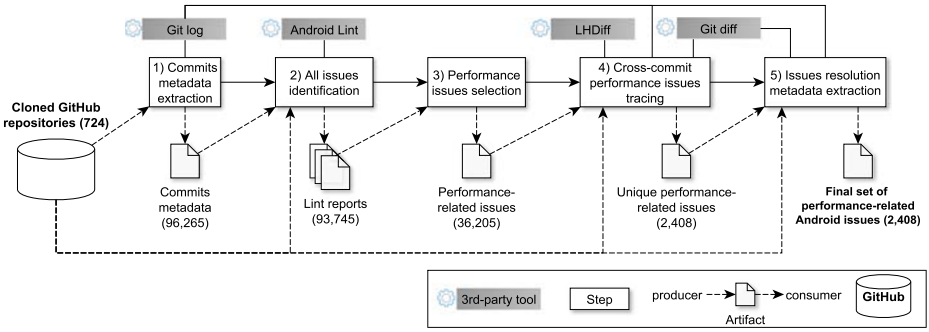


Fig. 5 Data extraction process

3.3 Data Extraction

Starting from the 724 GitHub repositories, we designed and implemented a tool chain for extracting the data for answering our research questions. As shown in Fig. 5, the tool chain is composed of five main steps, which have been implemented as a combination of Python scripts, shell scripts, and third-party tools. Each step will be discussed in details in the following.

Step 1 – Commits metadata extraction. The first step of our tool chain clones all 724 repositories locally in a dedicated machine. Then, for each cloned repository, we extract th change log, which contains SHA-1 hash, author, commit message, and timestamp. The extracted data is composed of 96,265 unique items (one for each commit within all GitHub repositories).

Step 2 – issues identification. In this step the tool chain iteratively (i) checks out each cloned GitHub repository at each of the 96,265 commits of our dataset and (ii) runs *Android Lint* on the app folder (as identified in Section 3.2) after each check out operation. This results in 96,265 runs of the *Android Lint* tool, which took a total of around 336 hours of uninterrupted execution time (∼2 weeks) on a machine equipped with an Intel i7 processor with 1.80GHz of frequency and 8Gb of memory. It is important to note that Android apps can target different versions of the Android Software Development Kit (SDK). The target SDK of each app must be known in order to properly resolve calls to the Android APIs. Therefore, before executing the *Android Lint* tool, we manually download all Android SDKs used in the apps of our dataset (from API levels 17 to 24) and locally stored them in a known location. Before each launch of the *Android Lint* tool, some preliminary action is necessary to be able to run *Android Lint* on arbitrary-developed Android apps. Firstly, our tool looks for the the app's Lint configuration file (i.e., the `lint.xml` file which may be in the folder containing the the app source code), checks if the Lint configuration file has the `abortOnError` set to `true` and removes it; this check is needed for allowing us to always fully run *Android Lint*, instead of stopping at the first error. Secondly, it removes all statements in the Lint configuration file for disabling any specific check; this check is needed because in some projects the developer may decide to explicitly disable some checks related to Android performance-related issues. Finally, our tool runs *Android Lint* by (i) resolving calls to the Android APIs by referring to the locally-downloaded SDK with the same version as the one specified in the Android manifest file of the current app and (ii) ignoring

all calls to external libraries, as we are interested in performance-related issues specific to the apps in our dataset. Starting from the 96,265 commits, *Android Lint* failed 2,520 times (2.61%) because of internal errors of the Android Lint tool (mainly due to peculiar configurations of the Android project). In these cases, the tool discards the considered commit and proceeds with the next one along the versioning history of the current app. This leads to the final set of 93,745 Lint reports, each of them stored as a separate HTML file.

Step 3 – Performance issues selection. In this step our we consider all Lint reports produced in step 2 and extract performance-related issues, as identified by *Android Lint*. To this aim, we developed a parser that takes as input the HTML file of a Lint report and automatically extracts the information we need about each identified issue contained in it. Specifically, the extracted information includes the following data items: (i) the category of the issue as defined in *Android Lint*,[18] (ii) the path to the source code file where the identified issue is located, (iii) the line number in the source code file where the issue is located, and (iv) the raw contents of the warning/error message. Finally, since we are interested in performance-related issues, we filter out all identified issues whose category is different from *performance*. The obtained dataset is composed of 36,205 performance-related issues found in 316 out of 724 apps.

Step 4 – Cross-commit performance issues tracing. After step 3 our set of performance-related issues contains a large number of duplicates. This is expected since an issue remaining in the code base for more than one commit appears multiple times among our set of 36,205 issues; more specifically, it appears exactly once for each commit where it is present. The main goal of this step is to remove those duplicates in order to have a set of unique Android performance-related issues, where each of them can possibly span more than one commit. A naive solution to this problem could have been to simply merge issues reported in subsequent commits which appear in the same Java file and in the same line number. However, a line of code can move during the lifetime of a GitHub project, both across different files (e.g., when a file is renamed or moved within the repository) and within the same file (e.g., some lines of code are added before the considered line of code). In order to correctly match potentially moving lines of code, we exploit the `git diff` tool and the *LHDiff* technique (Asaduzzaman et al. 2013a, b) in combination. Specifically, we use `git diff` for building the chain of versions of each file containing at least one performance-related issue, even when it is renamed or moved within the file system. We use `git diff` because it is accurate in identifying the renamed files in GitHub repositories and it is easy to integrate into our tool chain. We use LHDiff for tracking source code lines across two versions of the same file (Asaduzzaman et al. 2013a). We use LHDiff because (i) it is language-independent, thus applicable to Java source code files, (ii) it has been empirically evaluated and it its mapping process proved to be highly accurate, (iii) it is publicly available,[19] and (iv) it is distributed as a command-line tool, making it easy to integrate into our tool chain. Having a fully reconstructed tracing information about how each issue moves across and within source code files across commits allows our tool to identify those commits which are relevant for our study. Specifically, given an issue i_a of type a (e.g., *UseSparseArrays*) and $C_{i_a} = \{c_1, \ldots, c_n\}$ the set of commits in which i_a is present (i.e., still detected

[18] We take advantage of the fact that issues in *Android Lint* reports are tagged with a fixed set of categories (http://tools.android.com/tips/lint-checks) like performance, correctness, accessibility, usability, etc.

[19] https://muhammad-asaduzzaman.com/research

Table 2 Extracted data for each Android Lint performance-related issue type

Attribute	Type	Description
ID	String	the unique ID of the issue
repository	String	identifier of the GitHub repository of the issue in the form `author/repositoryName`
issueType	factor	type of the performance-related issue as identified by Android Lint (e.g., UseSparseArray, UseValueOf)
LintMessage	String	the warning/error message provided by Android Lint
introHash	SHA-1 hash	SHA-1 hash of the issue-introducing commit
introMessage	String	message of the issue-introducing commit
introTs	Integer	timestamp of the issue-introducing commit
preResHash	SHA-1 hash	SHA-1 hash of the issue pre-resolution commit
preResMessage	String	message of the issue pre-resolution commit
preResTs	Integer	timestamp of the issue pre-resolution commit
isResolved	Boolean	*true* if the issue is resolved, *false* otherwise
resHash	SHA-1 hash	SHA-1 hash of the issue-resolution commit
resMessage	String	message of the issue-resolution commit
resTs	Integer	timestamp of the issue-resolution commit
resLOC	Integer	number of Java LOCs changed in the issue-resolution commit

by the tool), we call c_1 the **introducing commit** and c_n the **pre-resolution commit**, The output of this step is composed of 2,404 unique performance-related issues. Each issue contains is represented by all the data items described in step 3 and the SHA-1 hash, message and timestamp of its introducing and pre-resolution commits.

Step 5 – Issues resolution metadata extraction. In this step we collect the metadata (i.e., SHA-1 hash, message, timestamp, and number of changed Java lines of code) related to the commit in which each performance-related issue has been resolved by developers (we called them **resolution commits**). In this context, by issue resolution commit we mean the commit immediately after the pre-resolution commit (i.e., the c_{n+1} commit in the discussion above). Moreover, if c_n is the last commit in the whole versioning history of the GitHub repository, it means that we reached the end of the lifetime of the project and the issue has never been resolved; in those cases, the issue is considered as **unresolved**, otherwise it is considered as **resolved** and we keep track of its resolution commit.

In Table 2 we summarize the data extracted for each Android performance-related issue. It will be used across the whole study and will be the base for the data analysis phase.

4 RQ_0 Results – Performance Issues Identified by Android Lint

4.1 Data Analysis (RQ_0)

For answering RQ_0, we present and discuss (i) the number of performance issues identified by *Android Lint* across all apps, (ii) the frequency and distribution of each type of performance issue across all apps, and (iii) the distribution of the number of occurrences of each

Table 3 The types of performance issues considered in this study

Issue name (priority)	Description
DrawAllocation (9/10)	It generally occurs due to allocating memory in a method that is invoked frequently to draw UI elements on the display. Allocating memory can be avoided by allocating the memory upfront, which leads to increased performance, thus potentially leading to a smoother user experience.
FloatMath (3/10)	It deals with the `FloatMath` data type; specifically, on modern devices the `FloatMath` Java object is slower than using `java.lang.Math` due to the way the JIT optimizes `java.lang.Math` objects.
HandlerLeak (4/10)	It is due to handler using the `Looper` or `MessageQueue` of the main thread. If the handler is not static, then the Android activity or service cannot be garbage collected, even after being destroyed. This may lead to memory leaks.
Recycle (7/10)	It occurs with the lack of calls to the `recycle()` method, when dealing with recyclable objects, such as `TypedArray`, `VelocityTracker`, etc. Calls to the `recycle()` method should be done after one of the above mentioned objects have been used, in order to make it reusable in the future.
UseSparseArrays (4/10)	It is mainly due to the use of `HashMap` instead of `SparseArray`. The Android framework promotes the usage of `SparseArray` over `HashMap` since it is assumed that sparse arrays are more memory efficient than `HashMap`, while not exhibiting large performance differences when dealing with hundred of items.
UseValueOf (4/10)	It is mainly due to direct calls to the constructor of wrapper classes (e.g., `new Integer(42)`), as opposed to calling the *valueOf* factory method (e.g., `Integer.valueOf(42)`). Calling factory methods is typically more memory efficient since common integers such as 0 and 1 share a single instance at run-time.
ViewHolder (5/10)	It occurs in the context of `ListViews`. When implementing a view Adapter, developers should avoid unconditionally inflating a new layout; if an available item is passed in for reuse, developers should try to use that one instead.
ViewTag (6/10)	Before the Android 4.0 version, `View.setTag(int, Object)` implementation stored the objects in a static map, where the values were strongly referenced. This implies that if the object references its calling context, the leak will happen from the context (which potentially may point to a large number of other objects within the app).
Wakelock (9/10)	It is due to failing to release a *WakeLock* properly, thus keeping the mobile device in high power mode, which decreases the lifetime of battery.

type of performance issues per app. Moreover, in order to better characterize statically-detectable Android performance issues, for each type of performance issue we provide and discuss an example of Java code exhibiting the issue. At the time of the experiment execution, *Android Lint* supports 9 types of performance issues,[20] they are presented in Table 3. Finally, to provide an overview of the apps' popularity, we use bar plots to show the range of number of downloaded apps from Google Play Store for each types of identified performance issue.

[20] http://tools.android.com/tips/lint-checks

We anticipate that in the remainder of this study we will not consider the *ViewTag* and *Wakelock* issues since they both occur only 3 times each within the whole dataset.

4.2 Results (RQ_0)

A total of 2,408 performance issues have been detected by Android Lint. Among them, 316 (43.64%) over 724 distinct apps suffered from at least one statically-detectable performance issue along their lifetime. Figure 6 presents the frequency of different types of performance issues in our dataset. Examples of code snippets containing the various types of performance issues are shown in Appendix A.

First, we can immediately notice how *Recycle* performance issues occur more frequently (550, 22.84%) than others. In order to optimize performances, collections such as *TypedArray*, VelocityTracker, Parcel or MotionEvent should be recycled after use, instead re-created again and also database cursor should be freed up after use. For example, as shown in Listing 2 (refer to Appendix A), *TypedArray preset_vals* should be recycled by a *recycle()* call (i.e., *preset_vals.recycle()*) for further reuse. The lack of a recycle could noticeably degrade the performance of the app.

Also *UseValueOf* issues type are quite frequent in our dataset (549, 22.79%). Since issues of type *UseValueOf* primarily deal with primitive types, we can conjecture that developers deem as negligible the potential performance improvement when resolving this kind of issues. Nevertheless, developers should take care of those issues since after a manual analysis we noticed that they may occur in the burst (Listing 3 in Appendix A for an example obtained from our dataset), thus potentially impacting the performance of the app in a noticeable manner.

UseSparseArrays (376, 15.61%) type of issues are third most common occurrence in our dataset. As discussed in Table 3, a performance degradation may occur when developers use a `HashMap` and the maps grows in an unpredicted manner. Listing 4 (see Appendix A) provides an example of `HashMap` Usage.

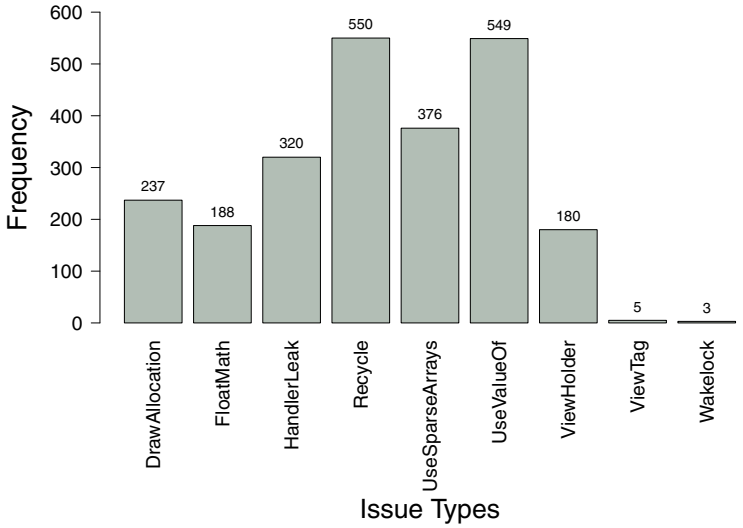


Fig. 6 Frequency of performance issue belonging to different categories

Then, there are potential *HandlerLeak* issues (320, 13.28%) in our dataset. The main consequences of this type of issue are memory leaks in the projects. To avoid this issue, developers should declare the handler as static. From the manual analysis in our dataset, we observed that in few cases developers intentionally resolved this issues, for example in Listing 5 (see Appendix A), developers mentioned in their comment (in the Java source code file) to declare static handler and for that, in the very next commit, they declared handler as static to get rid from potential memory leak issues. There are also many projects present in our dataset where developers do not declare the handler as static, which may lead to suffering from potential memory leaks as shown in Listing 6 in Appendix A.

Also there are issues of type *DrawAllocation* (237, 9.84%), can be found in various projects in our dataset. This is mainly due to allocation of memory when draw or layout operation is frequently invoked in a method. Thus it causes assigning memory each time whenever the function is called. From the manual analysis, we can presume that there are few projects where developers resolved these issues but still many projects suffering from these type of issues. For example, there are several new objects such as `new ArrayList` (at line 4 and 12) and `new pie (500)` (at line 18), are allocated in `onDraw()` (i.e., allocating objects during a draw operation), heavily lead to UI lag as shown in Listing 7 (see Appendix A).

There are (188, 7.80%) issues of type *FloatMath* in our dataset. Listing 8 (see Appendix A) shows an example of this type of performance issue.
In previous versions of Android, when working on floats, `android.util.FloatMath` was referred to ensure for performance reasons. However, on latest hardware doubles are equally quick as float (though they take more memory), and current Android versions, because of the way JIT optimizes `java.lang.Math`, FloatMath is in fact slower than using `java.lang.Math`.[21]

Issues of type *ViewHolder* (180, 7.47%) are more recurrent in our dataset after *FloatMath*. This type of issue primarily deals with the smoother scrolling of `ListView`. To show the `ListView` items, system has to draw each item separately. To reduce the number of `findViewById()` calls every time (when a list object has to draw), data from last drawn object can be reused (i.e., mainly by creating `ViewHolder` patterns). As shown in Listing 9 (see Appendix A), in `getView()` function every time a new object is draw (at line 5) followed by calling of `findViewById()` (at line 8) each time which may degrade the performance of app i.e., lag in smoother `ListView` scrolling. However, there are also certain cases in our dataset where developers specially implemented `ViewHolder` pattern class to avoid this issue.

ViewTag (5, 0.20%) are issues type that can be found very rarely in our dataset. These issues are related to the implementation of *View.setTag(int, Object)* and occurred in prior to Android 4.0. The consequences of this issue are leaks in the apps.

WakeLock (3, 0.12%) type of issues which are very less recurrent in our dataset. The WakeLock happened due to the failure to release a WakeLock properly that could keep the mobile device in high power mode and reduces the lifetime of the battery. There are many reasons of this phenomenon, such as failing to call *release ()* in all possible code paths containing *acquire()*,releasing the WakeLock in *onDestroy()* instead of in *onPause()*, and so on and so forth. Since this, it is a very crucial performance issue and developers do take care about the lifetime of battery while developing the app (i.e., releasing the wakelock

[21] Android floatmath documentation. https://developer.android.com/reference/android/util/FloatMath.

Table 4 Descriptive statistics for the number of statically-detectable performance issues per app (SD = standard deviation, CV = coefficient of variation)

Issue type	Max.	Mean	SD	CV
DrawAllocation	28	0.750	2.543	3.39
FloatMath	61	0.594	4.404	7.402
HandlerLeak	42	1.013	3.855	3.807
Recycle	75	1.741	6.114	3.513
UseSparseArrays	84	1.190	5.394	4.533
UseValueOf	171	1.737	10.478	6.031
ViewHolder	17	0.569	1.655	2.906

properly), and thus it may be one of the reasons why we have less *WakeLock* issues in our dataset.

Moreover, Table 4 reports the descriptive statistics for the number of statically-detectable performance issues per app. It can be clearly seen from Table 4, that *Recycle* issues occur quite frequently in our dataset (mean = 1.741 issues per app) followed by *UseValueOf* issues (mean = 1.737 issues per app) with higher standard deviation.

Also, issues of type *UseSparseArrays* and *HandlerLeak* are quite widespread in the analyzed apps, with an average 1.190 and 1.013 issues per app, respectively. Instead, issues of type *DrawAllocation* and *FloatMath* are relatively less frequent in our dataset, with 0.750 and 0.594 issues in each app respectively. *ViewHolder* issues are less frequent in our dataset (i.e., Mean = 0.569 issues). These issue type have also the lowest coefficient of variation than other issues.

Regarding the relationship between issue occurrence and popularity of apps in terms of number of downloads, issues of type *UseValueOf* were found 244 times in apps downloaded 10,000-50,000 times from Google Play Store; issues of type *UseSparseArrays* occurred 129

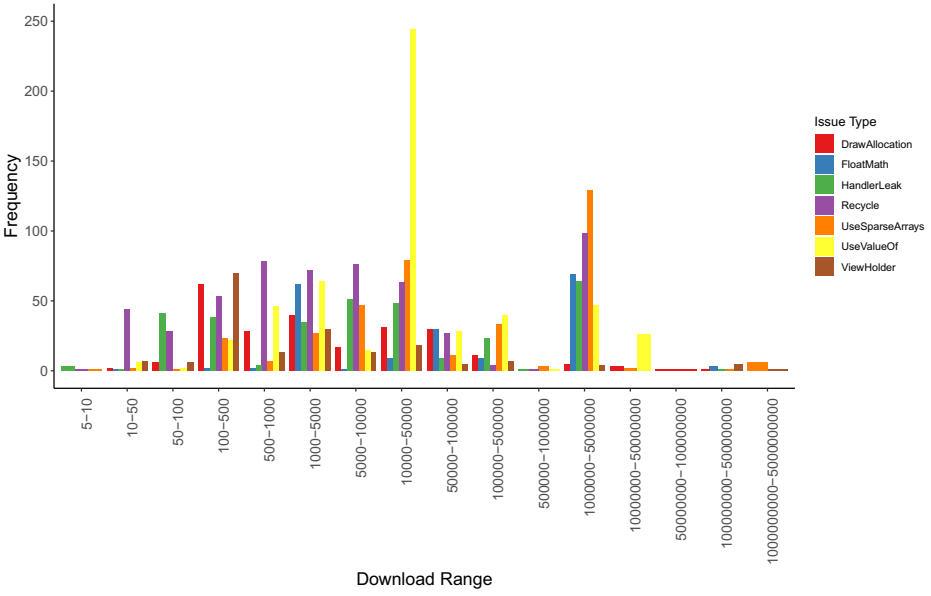


Fig. 7 RQ_0: Relationship between performance issue types and number of apps' downloads

times in highly-downloaded apps (1M-5M times). Figure 7 provides an overview of the different types of performance issue identified in apps having varying download ranges. One thing that immediately leaps to the readers' eyes is that, for apps having a very high number of downloads (greater than 5M) the issue frequency suddenly decreases. Although we do not have evidence of that, it is possible that such very popular apps undergo a more accurate quality check (e.g., code review), which may explain such a drop. We can also notice a very high frequency of *UseValueOf* issues for apps having a medium (1k-5k) number of downloads. Although this issue (calling constructors of wrapper classes instead of factory methods) may result in a waste of memory, we cannot tell why it happens for apps in that specific range of downloads.

Summary – RQ₀ – A total of 2,408 performance-related issues has been detected by Android Lint. *Recycle* issues are the most recurrent ones in our dataset (550, 22.84%), whereas *WakeLock* are the less recurrent ones (3, 0.12%).

5 *RQ₁* Results – Evolution of the Number of Android Performance Issues over Time

5.1 Data Analysis (*RQ₁*)

We answer RQ_1 by analyzing each of the 316 apps with at least one performance-related issue. For each app, we firstly reconstruct its versioning history by considering the sequence of all commits in its GitHub repository; then, for each commit we count the occurrences of any type of performance issue. The final result of this activity is a set of 316 plots, showing the evolution of the number of performance issues over the lifetime of each app, as well as the app size evolution in terms of LOC. The reason why we observe these two variables is because the former represents the main factor being investigated in this study, and the latter is a factor we need to control. This is because the growth of performance issues could be considered more problematic when it happens more rapidly than size increase. Examples of generated plots are shown in Section 5.2.

In order to characterize how performance issues evolve in Android apps, we perform a *qualitative study* on the plots and manually categorize them into relevant groups by applying the *open card sorting* technique (Spencer 2009). We perform the card sorting in two phases. First, we manually tag each plot with considerations about the presence of relevant evolution patterns, e.g., presence a spike, plateaus, sudden drops, etc. Then, we cluster identified patterns into meaningful groups with a descriptive title; each plot can exhibit more than one issue evolution pattern, i.e., it can belong to more than one group. To minimize bias, three researchers have been involved in the card sorting activity. Specifically, we randomly selected 100 apps from the dataset and the main author of this study categorized them. Then, the same 100 apps have been randomly assigned to the other two researchers involved in this study (50 apps each), who categorized them independently. The three emerging sets of categories were slightly different (see our replication package for their specific items) and have been collaboratively discussed and merged in order to agree on a final set of categories of evolution trends of Android performance issues.

Finally, the main author of this study categorized the 316 plots into the different possible evolution patterns, and the task was repeated by the second author for the first 158 apps and by the third author for the last 158 apps, so that for each app there were at least

Table 5 Categories of evolution patterns of Android performance issues, where P = the number of performance issues, LOC = lines of code, ∼ = irrelevant for the identification of the evolution pattern

Pattern	Description	P	LOC
STICK	*Sticky issues*, i.e., issues are injected abruptly and they remain in the app across several commits	step ↑	∼
REF	*Refactoring* of performance-related issues	step ↓	∼
BEG	Issues since the *beginning* of the project	stable != 0	∼
INJREM	*Injection and removal*, i.e., a relatively large number of issues is injected in the project, followed by a sudden removal	spike	∼
GRAD	*Gradual*, i.e.,, performance issues gradually occur during the app development process	=LOC	=P

two taggers performing the classification. In order to further reduce bias, the other two researchers re-applied the final set of categories to their initially assigned 50 apps and the level of agreement between each researcher and the first one has been assessed by means of the Cohen-Kappa statistics (Cohen 1968).

After the application of the open card sorting technique, we report and analyze the frequency of performance issue evolution patterns across all 316 apps. Moreover, from our analysis it also emerged that a large number of apps contain a combination of different categories of issue evolution patterns. In order to better investigate this aspect, we statistically assess their correlation by building a contingency table with rows and columns representing each issue evolution pattern and computing its Cramer's V coefficient (Rosenberg 2008). The Cramer's V coefficient is a well-known measure of association applicable to contingency tables involving two categorical variables and it is defined within the [0, 1] range, where 0 indicates no correlation and a value of 1 indicates perfect correlation.

5.2 Results (RQ_1)

To answer $\mathbf{RQ_1}$, we analyzed the evolution of the number of performance issues throughout the lifetime of each GitHub repository. The analysis of the 316 repositories of our dataset via the open card sorting technique described in Section 5.1 resulted in the identification of five different evolution patterns. Table 5 reports the emerging patterns.[22]

As discussed in Section 5.1, three researchers have been involved in the identification of the categories of issue evolution patterns iteratively and collaboratively. For each evolution pattern category, the Cohen Kappa index between the first author and the second and between the first author and the third is calculated. In all cases the Cohen Kappa is > 0.6, hence indicating a strong agreement.

Figure 8 reports the distribution of the issue evolution patterns across the 316 apps containing at least one occurrence of each performance issue. We can observe that STICK (209) and REF (124) are the most frequent performance issue evolution patterns, followed by BEG (111), INJREM (69), and GRAD (41). In the following, we discuss in detail about these evolution patterns.

[22] In the remainder of the study we will refer to performance issues as P and to lines of code as LOC.

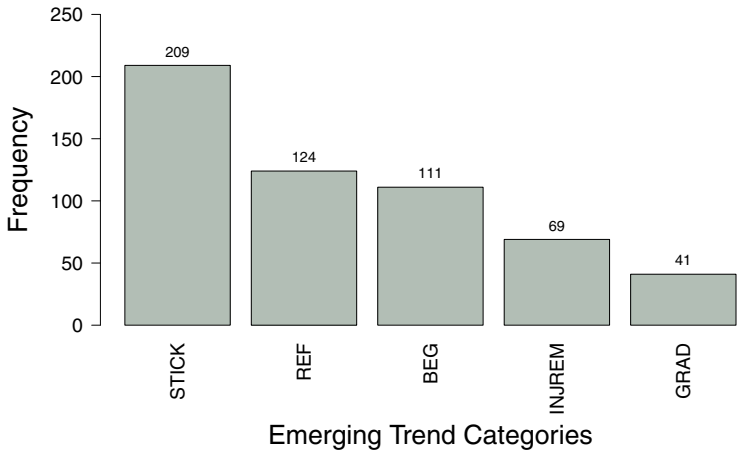


Fig. 8 Occurrences of performance issue evolution patterns across apps

STICK (Sticky issue). This pattern refers to issues which are introduced and remain in the project for several commits. In such a case, either the issue was not considered a serious concern (or even it was a tool's false positive), or developers had other priorities but resolving the performance issue. Figure 9 show the example of STICK patterns found in the analyzed apps. As it can be seen from Fig. 9, one issue (i.e., *HandlerLeak*) is introduced in the *alistairdickie/BlueFlyVario_Android project* and continue to remain in the system for many commits. As the project development progress further, another issue (i.e., *HandlerLeak*) is injected with the addition of new lines of code and then both these issues are stick to project for several commits (till the end of the project).

REF (Refactoring). This type of pattern indicates a possible refactoring action in the evolution of projects i.e., performance issues are resolved from the project with the increase or decrease of lines of code. In REF patterns, the number of performance issues dropped consistently, regardless of whether the overall LOC increased or decreased.

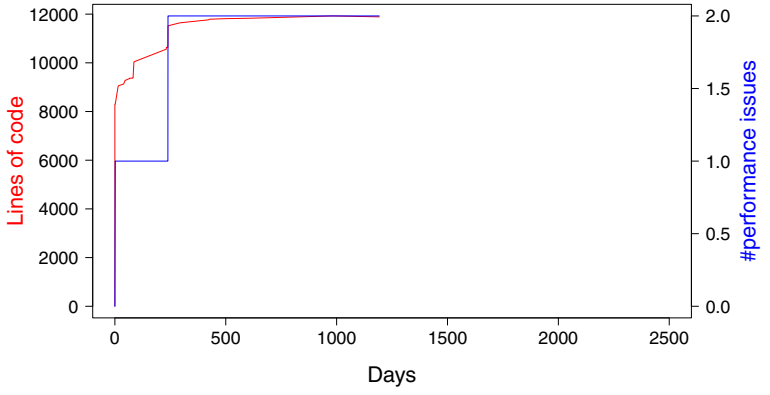


Fig. 9 alistairdickie/BlueFlyVario_Android - An example of STICK Evolution Pattern

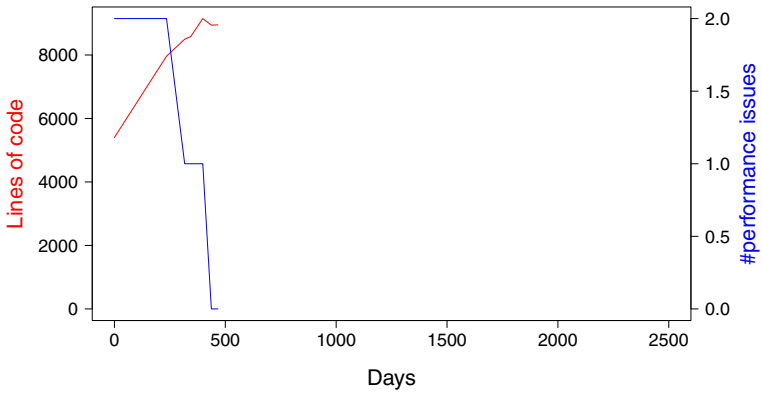


Fig. 10 xperia64/timidity-ae - An example of REF Evolution Pattern

From the manual analysis, we noticed that intentional refactoring done twice in the project xperia64/timidity-ae (as shown in the Fig. 10). Initially one *UseSparseArrays* issue was resolved by using `SparseIntArray` instead of `HashMap<Integer, Integer>`, whereas in the second refactoring ViewHolder Pattern was implemented inorder to resolved the *ViewHolder* issue.

BEG (issues since the Beginning). This pattern refers to cases in which performance issues which are present in the project since the beginning i.e., when the project was created.

As it can be seen from Fig. 11, related to the *offbye/ChinaTVGuide* app, this app contained 25 performance issues since the project's creation on GitHub, and they remained unaltered till the end of our observation period.

INJREM (Injection and removal). We identify the INJREM evolution patterns when a relatively large number of issues are injected in the project followed by a quick removal. We consider the resolution of a performance issue to be quick if it happens within two days from the introduction. Figure 12 show the example of INJREM pattern from our dataset. From Fig. 12 (pattern shown by the black arrow), it can be noted that in the app

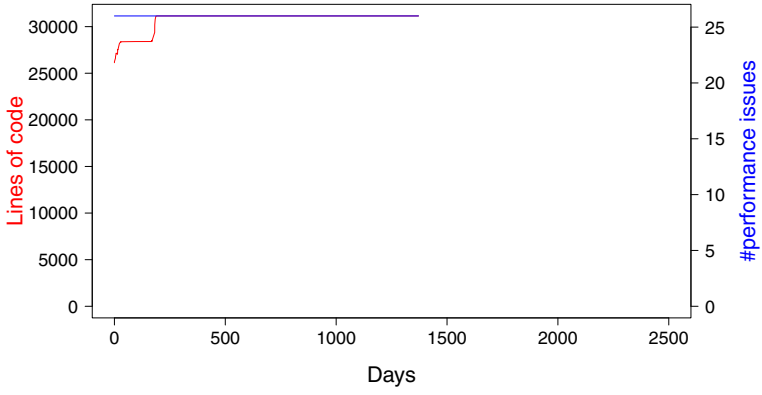


Fig. 11 offbye/ChinaTVGuide - An example of BEG Evolution Pattern

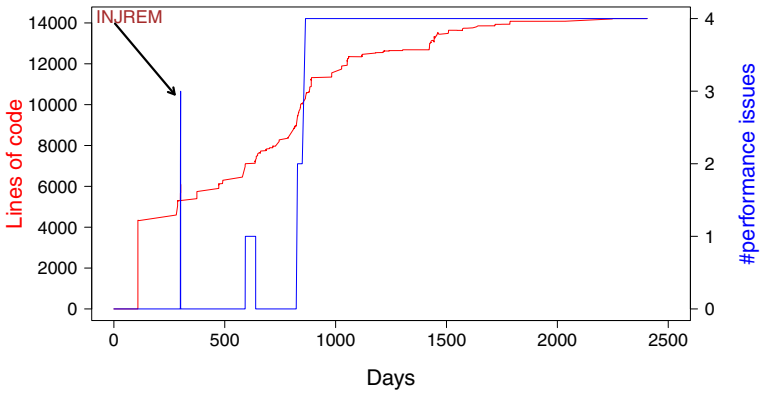


Fig. 12 mi9rom/achartengine - An example of INJREM Evolution Pattern

mi9rom/achartengine three issues were introduced (i.e., *UseSparseArrays*) and in the very short time interval, these issues were suddenly resolved. By manually inspecting the documented commit, we observed that the resolution was not done with intention of improving the performance of the app (i.e., the resolution was accidental).

GRAD (Gradual issues). In GRAD evolution pattern, performance issues gradually increase with the LOC or vice versa. In this type of evolution patterns, performance issues and lines of code grow or decrease altogether. In other words performance issues are an integral part of the app development process. As shown in Fig. 13, performance issues in the app *AlbertoCejas/GermanLearningUCA* (an example from our dataset) grow with the same rate of LOC. In other words, performance issues increase gradually with the gradual increase of lines of code.

Moreover, from the analysis of evolution patterns, we observed that many apps in our dataset which contain performance issues follow multiple co-occurrence patterns throughout the lifetime of their projects. To provide a quantitative indication of the association strength among patterns, we compute the Cramer's V coefficient, which measures the strength of association — varying between 0 to 1 — between two nominal variables. In our

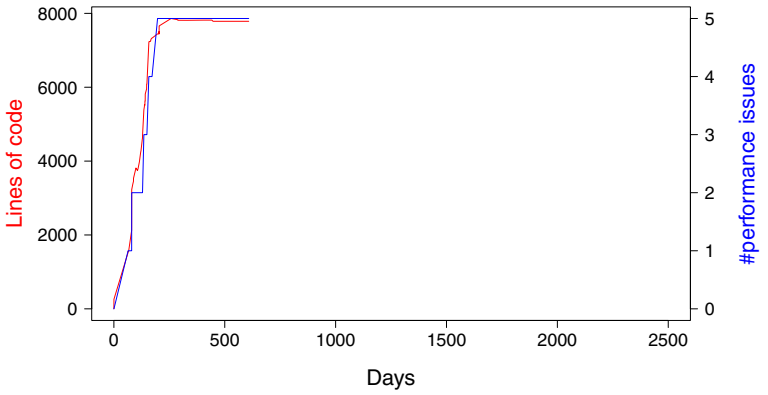


Fig. 13 AlbertoCejas/GermanLearningUCA - An example of GRAD Evolution Pattern

study, the computed Cramer's V coefficient value is 0.319 (which is low) meaning that there is a low association between the categories of emerging patterns.

> ***Summary – RQ_1*** – Five different evolution patterns emerged from the card sorting technique. STICK is the most frequent pattern in our dataset (209 occurrences), meaning that for 209 over 316 apps, issues tend to remain in the source code for several commits after their introduction. Moreover, while many patterns in our dataset co-occur, we did not find any pair of patterns exhibiting a high level of association.

6 RQ_2 – Performance Issues Remaining in Android Apps over Time

6.1 Data Analysis (RQ_2)

To answer RQ_2, we introduce two variables: RS_a^i and US_a^i. RS_a^i is defined as the number of resolved issues of type i (e.g., *DrawAllocation*) across the whole lifetime of app a. As discussed in Section 3.3, in this study a resolved issue is an issue which is present in the GitHub repository (for any number of commits up to commit c_j) and it is not present in the repository from commit c_{j+1} (in this case commit c_{j+1} is called the issue-resolution commit). Conversely, US_a^i is defined as the number of unresolved issues of type i within the lifetime of app a. Clearly, the sum of RS_a^i and US_a^i is equal to the total number of issues of type i for app a.

Given its exploratory nature, we answer RQ_2 by extracting and discussing the following information for each app a within our dataset and for each type of performance issue i: the ratio between the total number of performance issues of type i in a and its unresolved issue (resolved and unresolved issue are complementary to each other, therefore we only keep the unresolved ones).

Finally, we depict with bar plots the relationship between the type unresolved issue frequency and number of apps' downloads.

6.2 Results (RQ_2)

As shown in Table 6, *ViewHolder* issues tend to remain more than other types of performance issues across all apps in our dataset (101/180, 56.11%). Since *ViewHolder* issues primarily deal with the smoother scrolling of *ListView* items in Android apps, they can be

Table 6 Total number of Android performance issues and their subset of unresolved issues

Category	All	Unresolved (%)
ViewHolder	180	101 (56.11%)
UseSparseArrays	376	196 (52.13%)
DrawAllocation	237	119 (50.21%)
HandlerLeak	320	160 (49.00%)
Recycle	550	240 (43.64%)
UseValueOf	549	218 (39.71%)
FloatMath	188	52 (27.66%)
Total	2,400	1,086 (45.25%)

more problematic in terms of end users' experience. These issues tend to arise in the apps either due to lack of declaring a *ViewHolder* pattern, or not reusing the previously drawn items i.e., by reducing the number of calls to the *findViewById()* method.

Also *UseSparseArrays* issues (196/376, 52.13%) tend to remain unresolved in our dataset. For the purpose of efficient memory usage and less garbage collection, the Android platform promotes the use of `SparseArrays` instead of `HashMaps` for maps that contain up to a hundred values. From manual inspection, we assume that these issues remain unresolved due to unnoticeable memory improvements in terms of amount of memory (in bytes) allocated in the heap of the Java Virtual Machine.

The *DrawAllocation* issues (119/241, 50.21%) are the third less frequent issue type remaining unresolved in our dataset. Since draw operations are quite sensitive in terms of user-perceived performance, it is a bad programming practice to allocate memory (i.e., by declaring new instances) during draw or layout operations. This is one of the troublesome issue type that continues to remain alive in the apps.

The consequence of having *HandlerLeak* (160/320, 50.00%) issues is to have a memory leaks in the app, potentially leading to the usage of unneeded memory over long usage sessions. This type of issues can be resolved by declaring the handler as static.

Finally, we analyzed that *FloatMath* (52/188, 27.66%) issue type, which is resolved more frequently in our dataset as compared to other types of performance issues. Similar to *UseValueOf* issues, they also mainly deal with primitive data types and are resolvable in a relatively straightforward manner.

Concerning the relationship between of unresolved issues and apps' downloads, results shown in Fig. 14, highlight how `UseSparseArrays` issues remained unresolved 69 times in apps downloaded in the range of 1M-5M. Similarly to what found in RQ_0, also in this case we notice a difference (drop in the frequency of unresolved issues) for apps having a high number of downloads, above 5M. At the same time, it is also interesting to notice how *UseSparseArray* issues (see the orange bar) tend to exhibit a relatively high frequency

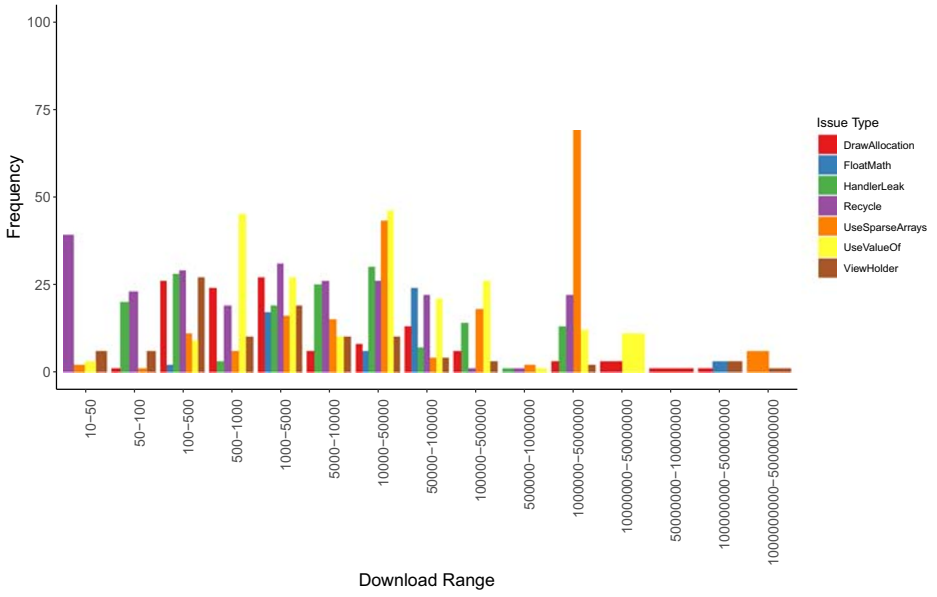


Fig. 14 RQ_2: Relationship between types of unresolved issues and number of apps' downloads

also for apps with a high number of downloads. This kind of issue suggests the use of `SparseArrays` instead of `HashMaps`, but it is possible that developers do not consider such an optimization as important given the size of the data they have to deal with in their apps.

> Overall, 45.25% of performance issues remain unresolved. *ViewHolder* issues are the ones remaining more in the app, even though they can be problematic in terms of end users experience since they primarily deal with a smoother scrolling of `ListView` items.
> *FloatMath* issues are the most resolved performance issues (52/188, 27.66%).

7 RQ_3 Results – the Lifetime of Android Performance Issues

7.1 Data Analysis (RQ_3)

We define the lifetime of an Android performance issue as the number of days between its issue-introducing commit and its issue pre-resolution commit. Intuitively, such a lifetime represents the time interval in which the issue is present in the source code of an app.

We answer RQ_3 in two phases. In the first phase we perform an initial exploration of the obtained lifetimes across all apps and report the summary statistics (together with box plots). In this phase, we order all issues according to their lifetime and trim outliers by removing the top 1% of all issues so to avoid potential issues related to repositories which have not been actively maintained in the last months.[23] In order to compare the different duration distributions, we apply the Kruskal-Wallis test (Kruskal and Wallis 1952) for each type of performance-related issue, followed by a Dunn post-hoc analysis (Dunn 1964). Since we are applying multiple statistical tests, in order to reduce the chance of Type-I error we correct the obtained p-values via the Holm p-value adjustment procedure (Holm 1979). The procedure sorts the n p-values obtained by the multiple comparisons in increasing order, and multiplies the smallest one by n, the second-smallest by $n - 1$, and so on (the largest one is left unchanged).

In the second phase, we aim at understanding whether Android performance issues exhibit some recurrent patterns in terms of resolution time. Specifically, we firstly reconstruct the cumulative distribution function (CDF) of the lifetime of each type of performance issues across all apps, then we plot it on a day-scale, and finally we investigate on whether each built CDF can be modeled using known statistical distributions. Specifically, we consider six known statistical distributions – *Cauchy, Exponential, Gamma, Lognormal, Normal and Weibull* – and assess to what extent each type of Android performance issues fits each of them. In total, we obtained 42 pairs (i.e., 6 statistical distributions $\times$ 7 types of Android performance issues) and assessed their fits. When fitting the CDFs to the known statistical distributions we follow a procedure similar to the one applied in (Di Penta et al. 2009), i.e., we (i) visually inspect each of the 42 plots showing together the CDF and known distribution in combination and (ii) statistically test how the known statistical distributions fit the CDFs by applying the Kolmogorov-Smirnov (KS) test to each pair. Specifically, we

[23] It is important to note that we performed the statistical analysis for RQ_3 both with and without outliers and the conclusions did not change

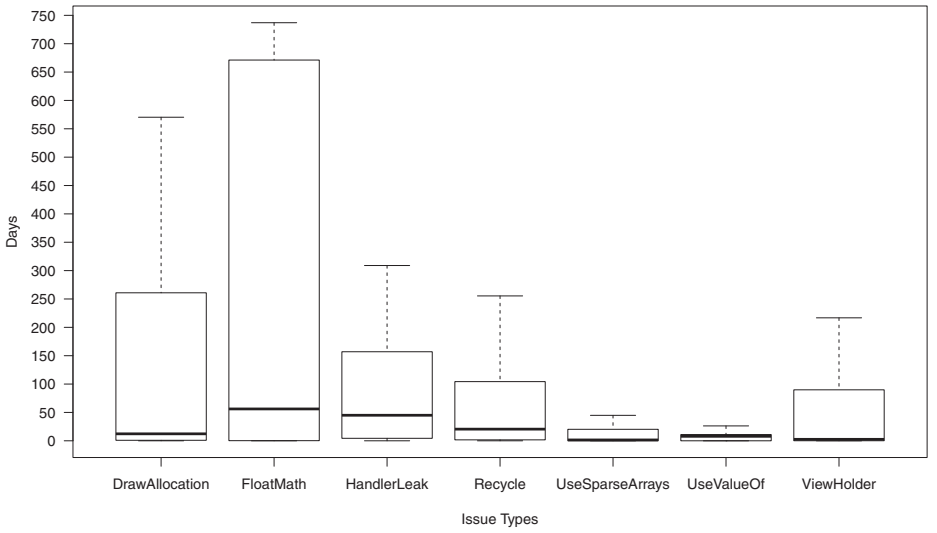


Fig. 15 Lifetime of each type of performance issues (outliers are not shown to help readability)

firstly estimate the distribution parameters of each CDF using the method of Maximum Likelihood, which maximizes the likelihood that the set of data used for the estimation can be obtained from the statistical distribution modeled with the estimated parameters. Then, we apply the Kolmogorov-Smirnov (KS), a non-parametric test that checks whether a distribution fits a given data (H_0 – there is no significant difference between the theoretical distribution and the actual data distribution). This means that every time the p-value of the applied KS test is greater than $\sigma = 0.05$, then the CDF fits the distribution. In Section 7 we report all CDFs with the best fitting known statistical distribution, all the obtained p-values, and a discussion of the obtained results. Finally, we use bar plots to depict the relationship between the type of frequently-resolved issues and the apps' number of downloads.

7.2 Results (RQ_3)

As discussed in Section 7.1, we answer $\mathbf{RQ}_3$ by analyzing the lifetime of each type of Android performance issue. As an initial exploration of the obtained results, Fig. 15 and Table 7 present the distributions and descriptive statistics of the lifetime of each type of performance issue across all apps.

Table 7 Descriptive statistics for the lifetime (in days) of each type of performance issues per app (SD = standard deviation, CV = coefficient of variation)

Issue type	Min.	Max.	Median	Mean	SD	CV
DrawAllocation	0.0008	864.6479	12.2779	139.0042	204.4615	147.0901
FloatMath	0.0009	736.9915	56.1576	293.8827	324.4333	110.3955
HandlerLeak	0.0001	1221.2477	45.0079	169.3382	289.2138	170.7905
Recycle	0.0001	961.0392	20.5430	105.8828	183.5087	173.3129
UseSparseArrays	0.0008	1147.3397	1.4995	95.7237	220.1935	230.0302
UseValueOf	0.0011	882.4600	7.8848	49.8220	130.8643	262.6637
ViewHolder	0.0022	833.1318	2.5965	107.1553	209.0570	195.0971

Table 8 Results of the Dunn's post-hoc analysis for comparing duration distributions (p-values are in parenthesis, statistically significant p-values are shown in bold and marked with *)

Col Mean-Row Mean	DrawAlloc.	FloatM.	HandlerLeak	Recycle	UseSparseA.	UseValueOf
FloatMath	–1.064105					
	(0.5746)					
HandlerLeak	–1.661472	–0.588860				
	(0.4348)	(0.8339)				
Recycle	0.110759	1.431713	2.214711			
	(0.4559)	(0.4567)	(0.1473)			
UseSparseA.	3.374780	4.729559	5.558179	4.159201		
	(0.0052*)	**(0.0000*)**	**(0.0000*)**	**(0.0003*)**		
UseValueOf	3.983202	5.572733	6.594601	5.317560	0.267410	
	(0.0005)**	**(0.0000*)**	**(0.0000*)**	**(0.0000*)**	(0.7892)	
ViewHolder	1.624728	2.636551	3.209208	1.790917	–01.248851	–1.555902
	(0.4169)	(0.0503)	**(0.0087*)**	(0.3665)	(0.5293)	(0.4191)

We can notice that the medians of lifetimes across issues types range from about 1.5 days (*UseSparseArrays*) to about 56 days (*FloatMath*), with very high standard deviations, which range from about 130 days (*UseValueOf*) to about 324 days (*FloatMath*). Having high standard deviations provides an indication that the lifetime of statically-detectable performance issues can vary across apps and projects.

Results of the pairwise comparisons performed by the Dunn's procedure are reported in Table 8.

Performance issues of type *DrawAllocation*, *FloatMath*, *HandlerLeak*, and *Recycle* have significantly longer lifetimes with respect to both *UseSparseArrays* and *UseValueOf*. Moreover, in our dataset *HandlerLeak* issues also have significantly longer lifetimes with respect to *ViewHolder* issues.

Table 9 reports the fitted distributions, their parameters, and the p-values of the KS test for each type of considered Android performance issue. The bold values represent the case when there is a possible distribution fitting (i.e., p-values $>$ 0.05). As shown in Table 9, *HandlerLeak* issues follow a *weibull* and *gamma* distribution with p-value $>$ 0.05, whereas *ViewHolder* issues fit the *weibull* and *lognormal* distribution. Furthermore, *DrawAllocation* issues fits the *gamma* distribution, *Recycle* issues fit the *weibull* distribution and *UseSparseArrays* issues fit a *lognormal* distribution. *FloatMath* and *UseValueOf* performance issues do not fit any considered distribution.

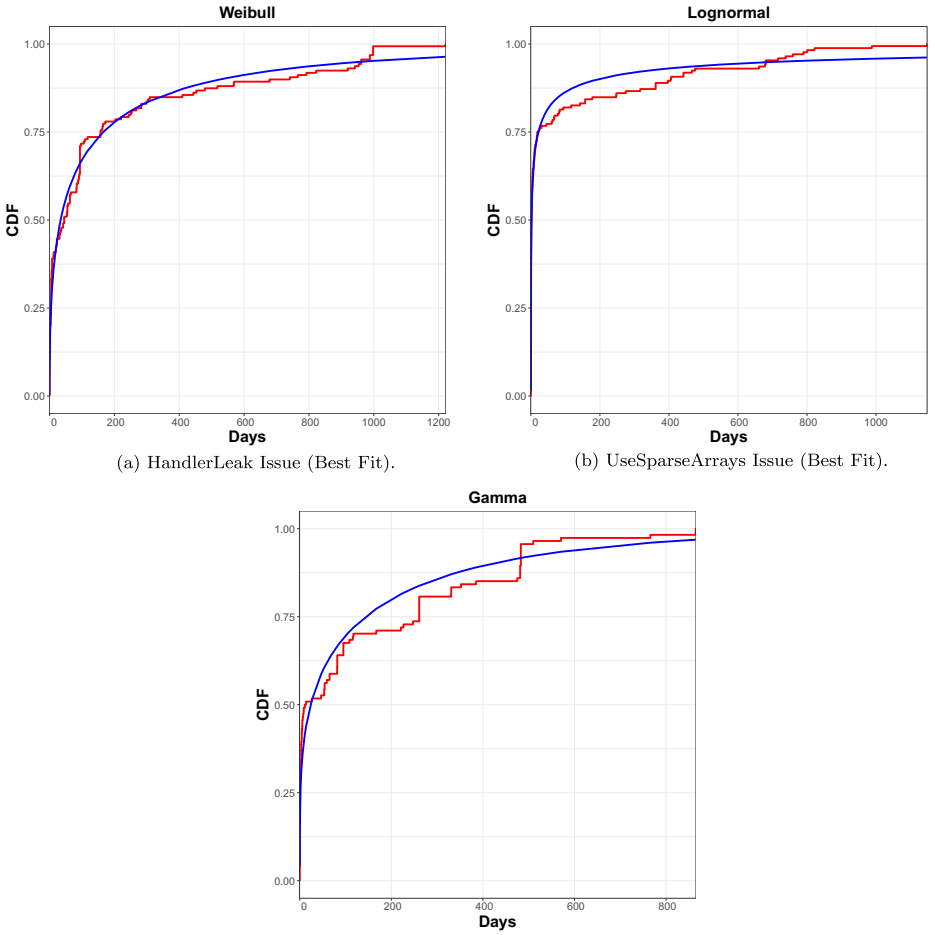
Figure 16 highlights the Cumulative Distribution Function (CDF) of few types of performance issues, in which the actual CDF is represented through a red line and the theoretical

Table 9 Results (p-values) of the KS test fitting the lifetime of different types of Android performance issue to different distribution models

Issue type	Norm	Exp	Weibull	Gamma	Lognorm	Cauchy
DrawAllocation	< 0.05	< 0.05	< 0.05	**0.17**	< 0.05	< 0.05
FloatMath	< 0.05	< 0.05	< 0.05	< 0.05	< 0.05	< 0.05
HandlerLeak	< 0.05	< 0.05	**0.33**	**0.13**	< 0.05	< 0.05
Recycle	< 0.05	< 0.05	**0.13**	< 0.05	< 0.05	< 0.05
UseSparseArrays	< 0.05	< 0.05	< 0.05	< 0.05	**0.23**	< 0.05
UseValueOf	< 0.05	< 0.05	< 0.05	< 0.05	< 0.05	< 0.05
ViewHolder	< 0.05	< 0.05	**0.28**	< 0.05	**0.29**	< 0.05

CDF is shown as a blue line. Figure 16 reports that the actual CDF for *HandlerLeak* issues follows the Weibull distribution. Instead, *UseSparseArrays* best fits the lognormal distribution and *DrawAllocation* fits a Gamma distribution. These plots indicate that those three types of issues tend to be resolved either (i) immediately (in all cases there is at least a 50% probability of resolving them within few days) or (ii) very late during the lifespan of the project (e.g., approximately 30% of *DrawAllocation* issues are resolved after 200 days).

We also analyzed the relationship between the type of issues frequently resolved and the apps' number of downloads. Results are shown in Fig. 17, and indicate that `UseValueOf` issues exhibit a high number of resolutions (198 times) in apps having a relatively medium number of downloads (10k-50k). While, as we discussed in RQ_0 (Section 7) we noticed a high number of issues of this type being introduced, we can also see that they are removed after a while.



(a) HandlerLeak Issue (Best Fit).

(b) UseSparseArrays Issue (Best Fit).

(c) DrawAllocation Issue (Best Fit).

Fig. 16 Empirical and theoretical CDFs

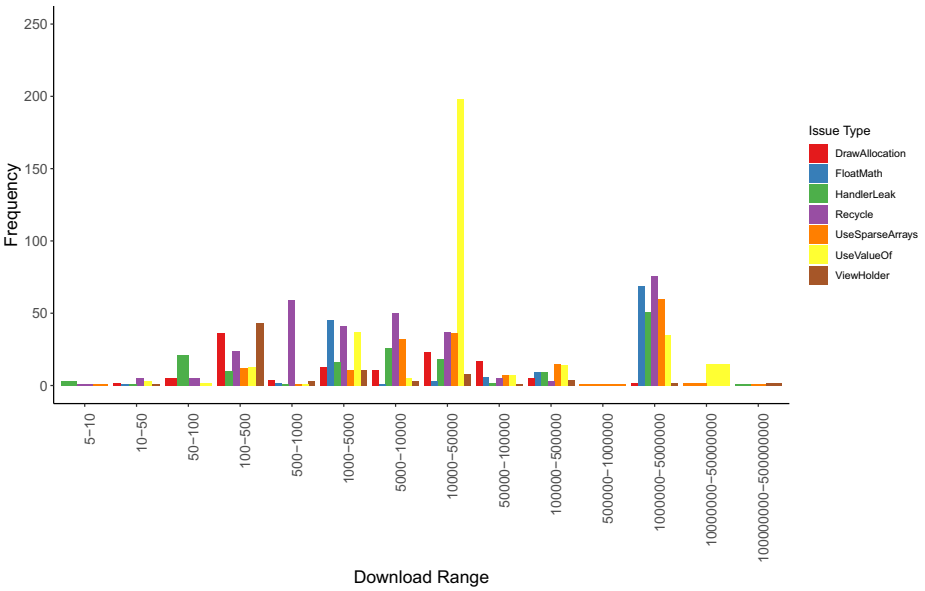


Fig. 17 RQ_3: Relationship between types of resolved issues and number of apps' downloads

Summary – RQ3 – Statically-detectable Android performance issues tend to remain in the project for a relatively long period. On average (median) they are resolved after more than 137 (21) days during the lifetime of the whole project.
When resolved, there is a 50% probability that a statically-detectable Android performance issue is resolved before 2 months after its introduction in the app's repository.

8 *RQ4* Results – Documented Resolutions of Android Performance Issues

8.1 Data Analysis (*RQ4*)

RQ_4 is about *documented* resolutions of Android performance issues. In this context, a documented resolution of a performance issue is a special type of issue resolution where the developer performing the resolution is consciously improving the performance of the app. The starting point of our analysis is the set of issue-resolution commits, which we collected during the data extraction.

In this phase, we consider GitHub *commit messages* as indicators of the actual documented intention of the developer. Then, for each type of Android performance issue, we define a dedicated set of terms for identifying which commit messages are dealing with that issue. Clearly, the identification of the sets of terms is a key factor for the success of this phase. We follow a semi-systematic approach for the identification of the terms related to each Android performance issue: (i) we extract an initial set of terms based on the name of each performance issue (e.g., *DrawAllocation* is composed of the keywords `draw` and `allocation`) and (ii) we manually extract relevant terms from the

Table 10 Regular expressions for identifying documented Android performance issues

Type of issue	Keywords
DrawAllocation	[Dd]raw*, [Aa]lloca*, [Ll]ayout, UI, [Mm]emory
Recycle	[Cc]ursor*, [Rr]ecycle*, [Tt]ypedArray*, [Vv]elocityTrack*
ViewHolder	[Vv]iewHolder*, [Aa]dapter*, [Ll]istView, [Ss]crol*
HandlerLeak	[Hh]andler*, [Ww]eakReferenc*, [Mm]essageQueue*, [Ll]eak*
UseSparseArrays	[Ss]parseArray*, [Hh]ashMap*
UseValueOf	[Vv]alue*, [Dd]ouble, [Ll]ong
FloatMath	[Ff]loatMath
Miscellaneous	[Ll]int, [Ww]arn*, [Pp]erf*

description of each type of performance issue in the official *Android Lint* web page. For example, the description provided by *Android Lint DrawAllocation* is provided below and it leads to the identification of the following terms: $< Draw, Allocate, Layout, User\ Interface, UI, Memory >$.

> You should avoid allocating objects during a drawing or layout operation. These are called frequently, so a smooth UI can be interrupted by garbage collection pauses caused by the object allocations. The way this is generally handled is to allocate the needed objects up front and to reuse them for each drawing operation. Some methods allocate memory on your behalf (such as Bitmap.create), and these should be handled in the same way.

By applying the afore mentioned procedure to each type of Android performance issue, we identify the regular expressions shown in Table 10.

It can also be noted from Table 10 that in last row of the table, we define an additional set of regular expressions containing few more general performance-related terms. We add this set of regular expressions in order to cover also those cases where the developers are addressing performance issues, but are not referring to any specific *Android Lint* check.

To answer RQ_4, we present and discuss (i) the frequency of documented resolutions of Android performance issues within the whole dataset of Android apps, (ii) the distribution of documented resolutions of performance issues across the seven types of Android performance issues, and (iii) the distribution of the *code churns* associated to each of the 1,314 PRI-resolving commits across each type of Android performance issue. Code churns refer to the total number of changed lines of code in a commit (either added, removed or updated) (Munson and churn 1998). In this study, we rely on code churns because (i) it is one of the most used metrics for representing the change volume between two versions of the same system (Munson and churn 1998), (ii) it can be considered as a relatively good estimator for the development effort devoted to a GitHub commit, and (iii) it can be extracted automatically with low computational effort, and (iv) the `git log` command can compute it out of the box.[24]

Also, to better explain the results, we provide an example of solution for each type of Android performance issue. This allows us to build a minimal catalog of reusable solutions,

[24] https://git-scm.com/docs/git-log

which can be used by Android developers for better understanding how to resolve *Android Lint* warnings in their projects. We select the solutions in two steps. First of all, we randomly select five commits for each type of Android performance issue, where a developer documented a performance-related change. Then, we manually inspect the changes in the Java source code in each identified commit and select the most recurrent and representative one (also based on the description of the corresponding *Android Lint* check). In Appendix B we present and discuss in details all the representative solutions we identified. We believe that developers can exploit this emerging collective knowledge for solving the performance issues of their mobile apps in a more effective manner.

Finally, we use barplots to depict the relationship between the types of documented issues and the apps' number of downloads.

8.2 Results (RQ_4)

Starting from the 1,314 commits where performance issues have been resolved (either intentionally or not), we identified the subset of commits whose commit message mentions Lint-related performance issues via the keyword-based strategy described in Section 3.3. For each type of Android performance issue, Table 11 reports the number of commits in which (i) developers explicitly document the resolution of a given type of Android performance issue (second column) and (ii) developers just generically mention that they resolved a performance issue (third column). We add this Miscellaneous set of keywords in order to cover also those cases where the developers are addressing performance issues, but are not referring to any specific Android Lint check (these Miscellaneous set of keywords are shown in the last line of Table 10).

Firstly, we observe that the resolution of every type of performance issue has been explicitly documented at least more than once, summing up to a total of 143 (10.88%) commits out of 1,314. Table 11 indicates that *Recycle* issues are more often documented (i.e., 48 times), followed by *FloatMath* (20 times), *HanderLeak* (17 times), and *ViewHolder* (16 times). This may be considered as an indications of the extent to which developers are aware of issues, which could be possibly related to the actual usage of static analysis tools like Android Lint.

There are 27 commits notes identified after applying the miscellaneous regular expressions (see Table 10). Among them, 9 commits are related to *Recycle* and *UseValueOf* issues, respectively.

Table 11 Number of documented performance issue resolutions

Issue type	#Commits	#MiscellaneousCommits	Total
DrawAllocation	12 (8.4%)	1 (0.7%)	13 (9.1%)
FloatMath	20 (13.9%)	2 (1.4%)	22 (15.3%)
HandlerLeak	17 (11.9%)	1 (0.7%)	18 (12.6%)
Recycle	48 (33.6%)	9 (6.3%)	57 (39.9%)
UseSparseArrays	2 (1.4%)	4 (2.8%)	6 (4.2%)
UseValueOf	1 (0.7%)	9 (6.3%)	10 (7.0%)
ViewHolder	16 (11.1%)	1 (0.7%)	17 (11.9%)
Total	116	27	143

In Table 12 we provide an example of commit for each type of performance issue and three examples of commits matching the generic regular expression. We also check how many of the 143 commits are exclusively related to the resolution of performance issues, and how many are tangled commits also related to other changes (e.g., the implementation of a feature, etc.). It is interesting to note that many of such commits refer only to performance issues resolution (i.e., 117 out of 143 commits). This suggests that, when documenting Lint-related resolutions in their commit messages, Android developers tend to do dedicated "issue resolution sessions", which are 100% focused on resolving Lint-related performance issues.

When looking at the minimum number of lines of code to resolve issues (see Table 13), the amount of code to be written to resolve some performance issues is fairly limited. For example, *Recycle*, *UseValueOf*, *FloatMath*, and *UseSparseArrays* can be resolved with 1, 2, 2, and 5 lines of code, respectively. This is because these kinds of problems mainly deal with the usage of primitives data types and thus their resolution can be performed with a very limited number of changes. For example, a *Recycle* issue can be resolved by closing a cursor `c` by invoking the *c.close()* or through recycling a TypedArray `t` i.e., by invoking

Table 12 Examples of commits with documented performance issue resolution

Category	Repository	Commit ID	Commit message
DrawAllocation	mchow01/FingerDoodle	15c6432	▲ *Fixed NullPointerException at FingerDoodleView.java line 67 edu.cs.tufts.mchow. FingerDoodleView.onDraw*
FloatMath	almalence/OpenCamera	b232324	▲ *Partially fixed issue with preview on Android 6 in camera2 mode. Fixed nexus naming in CC. Changed depricated FloatMath to Math.*
HandlerLeak	stdev293/battery-waster-android	7212e95	▲ *static handler to avoid potential memory leak*
Recycle	uberspot/AnagramSolver	be502aa	▲ *close cursors after using them.*
UseSparseArrays	pocmo/Yaaic	fb90f72	▲ *Use SparseArray instead of HashMap for in-memory server storage.*
UseValueOf	AmrutSai/sikuna	696873b	▲ *Made changes based on recommendations from the lint tool...*
ViewHolder	chaosbastler/opentraining	3052768	▲ *Implemented ViewHolder-Pattern for ExerciseImageListAdapter (used for CreateExerciseActivity). Awesome performance improvements.*
Other(UseValueOf)	Anasthase/TintBrowser	d6f86cd	▲ *Correct lint warnings.*
Other(Recycle)	shlusiak/Freebloks-Android	ec65540	▲ *fix lint warnings*
Other(ViewHolder)	LukeStonehm/LogicalDefence	9312039	▲ *FIXED: #re-use view if already exists, this will increase performance a little*

Table 13 Descriptive statistics for the LOCs for resolving each type of performance issue (SD = standard deviation, CV = coefficient of variation)

Issue type	Min.	Max.	Median	Mean	SD	CV
DrawAllocation	16	170	170	138.8	59.9	0.4
FloatMath	2	25	21	19.4	7.5	0.4
HandlerLeak	33	936	810	639.8	303.5	0.4
Recycle	1	598	578	328.2	277.5	0.8
UseSparseArrays	5	1,313	21	446.3	671.4	1.5
UseValueOf	2	148	4	53.3	65.6	1.2
ViewHolder	3	126	54	49.4	29.5	0.6

the *t.recycle()* method. Moreover, issues such as *UseValueOf* can be resolved by calling the `valueOf` factory method instead of directly calling the constructor of a wrapper class like `Integer(int)`.

With a minimum number of 16 and 33 changed LOCs, *DrawAllocation* and *HanderLeak* issues seem to be not trivially resolvable. For example, resolving an *HandlerLeak* usually implies the creation of a new static handler and setting a weak reference to it.

Figure 18 depicts the relationship between the number of apps' downloads and the type of documented resolved issues. It can be noticed that issues of type *Recycle* are resolved and documented a high number of times (38 times) in apps with a relatively low number of downloads (500-1,000). Instead, issues of type *FloatMath* are resolved and documented 20 times in highly-downloaded apps (range 1M-5M). As described in the Android Lint documentation, the use of `android.util.FloatMath` was recommended for performance issues in old versions of Android. Nowadays, for code addressing newer versions of Android

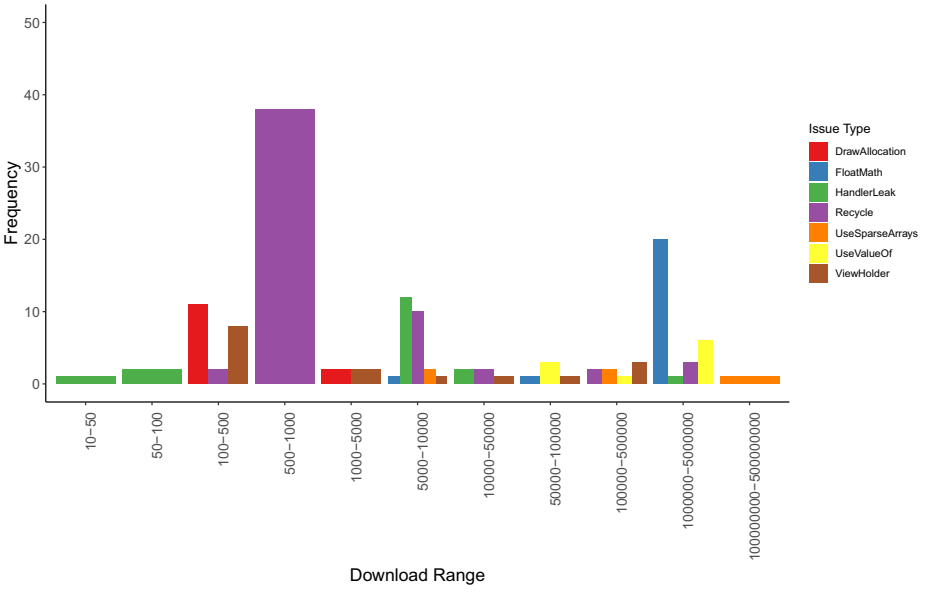


Fig. 18 RQ_4: Relationship between types of documented and resolved issues and number of apps' downloads

(Froyo or above) it is recommended to use `java.lang.Math` instead. It is possible that even popular apps did not update yet towards a better performance optimization for newer Android versions.

> ***Summary – RQ4*** – A total of 143 (10.88%) commits out of 1,314 document the resolution of PRIs. Out of those 143 commits, the resolution of *Recycle* issues is the most documented (57, 39.9%) and the resolution of *UseSparseArrays* issues is the least documented (6, 4.2%). A catalog of manually-extracted resolutions of performance-related issues is provided in Appendix B.

9 Discussion

In the following, we firstly provide a detailed analysis about how the results of the research questions of this study are linked together (Section 9.1). Then, we discuss the implications that the results of our study have for developers (Section 9.2) and researchers (Section 9.3).

9.1 Summary of the Study Results

Table 14 presents an overview of the main results discussed in the previous sections. Specifically, for each type of considered performance issue (first column) we report: its priority as reported in the Android Lint documentation (second column), its frequency in our dataset (third column), the portion of resolved issues (fourth column), its median lifetime (fifth column), the median of the lines of code for resolving it (sixth column), and median of documented resolutions (seventh column).

By looking at the combined results we can draw a number of interesting insights. Firstly, *DrawAllocation* has the highest priority among the considered performance issues, but it is resolved in roughly half of its occurrences (similarly to the resolutions of the other types of issues). We argue that this phenomenon can be explained by the relatively high number of lines of code needed for resolving it (170) and by the fact that it is difficult for developers to precisely assess when drawing or layout operations are taking place at run-time. *DrawAllocation* issues may be an impactful scientific target for researchers working on automatic refactoring tools because of the high priority of this type of issue, which can be considered as a proxy of the severity of the impact that this type of issues can have on the performance of the app.

Table 14 Combination of the obtained results

Issue type	Linter priority	Frequency (%)	Resolutions (%)	Lifetime (median)	LOCs (median)	Documented (median)
DrawAllocation	9/10	237 (9.84%)	118 (49.79%)	12.2779	170	13
Recycle	7/10	550 (22.84%)	310 (56.36%)	20.5430	578	57
ViewHolder	5/10	180 (7.47%)	79 (43.89%)	2.5965	54	17
UseValueOf	4/10	549 (22.79%)	331 (60.29%)	7.8848	4	10
UseSparseArrays	4/10	376 (15.61%)	180 (47.87%)	1.4995	21	6
HandlerLeak	4/10	320 (13.28%)	160 (50%)	45.0079	810	18
FloatMath	3/10	188 (7.80%)	136 (72.34%)	56.1576	21	22

Recycle is the most frequent type of performance issue (550 occurrences) and they are resolved in more than half of the cases (310). In principles, resolving this type of issue means changing only one line of code, i.e., adding a call to the `recycle()` method of the resource being used; however, the median number of lines of code for resolving *Recycle* issues is much higher (578 LOCs). At the time of writing, we do not have a precise explanation of this phenomenon and further investigation is needed.

We can observe that *UseValueOf* issues are resolved in a relatively short period (about 8 days) compared to other types of issues. This phenomenon may be explained by the fact that the median number of lines of code for resolving those issues is only 4. Indeed, by checking the official documentation of Android Lint, *UseValueOf* issues can be resolved simply by calling the `valueOf()` method of wrapper classes (e.g., `Integer`) instead of directly calling their constructor. Nevertheless, *UseValueOf* is the most recurrent type of performance issues in our dataset (549 occurrences) and their resolution is documented only in 10 cases. We argue that since *UseValueOf* issues are mostly about wrapper classes of primitive types, developers perceive this type of issues as not impactful with respect to the overall performance of the app.

While resolving *FloatMath* issues requires a relatively small effort in terms of lines of code (median = 21), their lifetime is the highest across all performance issues (median = 56 days). One possible reason for this type of issue to remain for such long periods is that Android Lint rates it with a very low priority (3/10), therefore developers tend to defer their resolutions more than they do for other types of issues.

Finally, we notice that, in general, the considered performance issues tend to be unresolved and persist for long periods. We argue that this phenomenon can be explained by two facts: (i) in general the priority of the considered issues is quite low (only *DrawAllocation* and *Recycle* have 9/10 and 7/10 as priorities, respectively, while the others have a priority level lower or equal to 5/10) and (ii) by default Android Lint only raises a *warning* when it detects one of the issues in the source code, leaving to the developer the choice to either solve or ignore the detected issue without blocking his/her development flow. Under this perspective, as we will also discuss in Section 9.3, it will be fundamental to empirically characterize the actual impact of Android Lint issues on the overall performance of the app. The results of such an assessment will help developers in taking better informed decisions about whether and when Android Lint performance issues should be managed or can be safely ignored to have Android apps with good-enough performance levels.

9.2 Implications for Developers

Based on the obtained results, in the following we summarize how such results could guide developers in better handling Android performance issues.

D1 – Developers are generally aware of performance issues detected by Android Lint, but there is space for improvement The results of R_0 are revealing that in general developers are not injecting an extremely high number of performance issues in their apps ($min = 0, max = 171$ issues per app). Nevertheless, within the limits of the Android Lint accuracy, the 2,408 performance issues we identified in this study can be seen as *missed chances* for improving the performance of Android apps.

Some of the detected issues are not only about performance, but are indicators of (i) poor programming practices (e.g., *UseSparseArrays*), which may hinder the overall maintainability of the app and (ii) memory leaks (e.g., *HandlerLeak*), which may potentially lead to the OS forcefully closing the app to recollect all the (mis-)used resources. Those are risks

that today's Android developers cannot afford in a crowded and fiercely competitive market as the Google Play store.

D2 – Do not treat all types of Android performance issues equally, but (re)use previous experience to prioritize and fix them Indeed, Android Lint checks are organized in different levels of priority and severity in order to guide the developer in prioritizing them. Also, when answering RQ_4 it emerged that *resolving Android performance issues demands different levels of effort*. For example, a *Recycle* issue can be simply resolved by a call to the *recycle()* method of the used recyclable resources (e.g., `TypedArray`); such a call can heavily improve the performance of the app, since it frees the potentially large resource before the execution of the garbage collectors. Differently, the resolution of a *HandlerLeak* issue is usually implemented by (i) creating a static inner class for the handler, (ii) having a weak reference in the outer class pointing to the outer class, and (iii) always using the weak reference when referring to the outer class. One possibility, to support developers, is that researchers (see Section 9.3) develop linters that prioritize warnings based on some knowledge. At the same time, as a lesson for developers, this paper attempted to distill and discuss a catalog of solutions for the various kinds of issues, based on what developers have done in the studied apps. Such a catalog (Appendix B) is discussed in our RQ_4 (Section 8).

Generally speaking, we advice developers to *establish different priority levels to different performance issues* depending on app's users' needs, project characteristics, and available resources and to build a prioritization model according to them. In this context, variations of the Weighted Shortest Job First (WSJF Leffingwell, 2010) model may be a good starting point. At the same time, developers should, (once again, possibly with the help of environments developed by researchers) build a knowledge base of previously adopted solutions, in order to apply them when appropriate.

D3 – Performance issues may be harmless when they occur in isolation, while they can be particularly concerning when they occur in combination . If we look at each Android performance issue in isolation, its overhead at run-time may be minimal in terms of computational resources demanded by the app. However, in this study we also observed that (i) in many cases issues tend to accumulate over time (e.g., STICK is the most recurrent evolution pattern) and (ii) apps can exhibit many performance issues at the same time (e.g., the app ChinaTVGuide exhibited 25 performance issues for more than 4 years).

Developers can alleviate these risks by *continuously monitoring* the number and types of Android performance issues in their code base during the whole project. This practice can also help in mitigating the well-known problem that developers are less likely to fix linter warnings on legacy code (Ayewah et al. 2008; Habchi et al. 2018). Indeed, if the monitoring (and corresponding resolution) of performance issues is performed continuously along other development activities, then it will be unlikely that the code base will accumulate a high number of issues, thus avoiding the need to fix them on previously developed code. In this context, one concrete possibility is to configure linters (e.g., in a Continuous Integration pipeline) so that they warn developers, by failing a build, when too many potential performance issues have been introduced in a commit, or in a sequence of consecutive commits. Last, but not least, it can be advisable to complement linters with performance regression tests.

The five evolutionary patterns of statically-detectable Android performance issues (e.g., issue-rich feature, injection, and removal, etc.) are quite heterogeneous and exhibit different characteristics, and it is difficult to track them by looking at one snapshot of the source

code of the app at a time (e.g., gradual or sticky issue). The *identified evolution patterns can be used by Android developers as a common, shared, and aggregated viewpoint* for guiding maintenance activities and keeping under control the health of their apps from the perspective of performance issues. The integration of the tool chain discussed in Section 3.3 and visual dashboards may be an invaluable instrument for developers for readily identifying the manifestation of dangerous evolution patterns (e.g., issue-rich feature), and thus take immediate action in response to them.

9.3 Implications for Researchers

In the following, we phrase the implications for researchers as Perspective Research Questions, labeled as PRQ — which can be addressed in future research work.

PRQ_1 – Why some performance issues tend to remain in Android apps? In our study, Android performance issues tend to remain for many days within the repository; the average duration ranges from 53.95 days for *UseValueOf* to 293.9 days for *FloatMath* issues. We suspect that those long durations are due to the fact that performance issues raised by Android Lint are deemed to be not dangerous by developers.

We still do not have evidence about whether this perception is true or not, but objectively investigating *how developers perceive the issues raised by Android Lint* is a relevant research direction. An initial investigation is reported in Habchi et al. (2018), where the main causes for developers not to use Android Lint have been empirically extracted via interviews. There, the top three beliefs against the use of Android Lint for performance purposes are: (i) that performance issues should be managed reactively (i.e., until someone complains), (ii) that static analysis is not suitable for performance, and (iii) that Android Lint performance checks are irrelevant.

In addition to the common wisdom mentioned above, other factors strictly related to the tool itself may influence the lack of resolutions of Android performance issues. For example, given its static nature, sometimes Android Lint can produce false positives (i.e., raising warnings when actually there is no issue), thus negatively impacting developers' trust in it (Bessey et al. 2010). Also, even just the wording of the error messages shown to the developer may affect the tool's adoption; indeed, if developers find the error messages as confusing, then they deem the raised error as false (Bessey et al. 2010). While this is not entirely surprising, as it confirms findings coming from studies on general-purpose linters (Couto et al. 2011; Kim and Ernst 2007; Spacco et al. 2006; Wedyan et al. 2009), it suggest that, perhaps, more advanced recommenders are needed. For example, as it has been done for bug fixes (Tufano et al. 2019), it could be interesting to learn from past changes to prioritize warning resolution. Finally, developers may tend to perceive the resolution of Android performance issues as demanding too much effort for the obtained gain. This calls for the next two research questions (PRQ_2 and PRQ_3).

PRQ_2 – To what extent it is possible to automatically refactor performance issues in Android apps? Almost half of the apps in our dataset (43.64%) exhibit at least one statically-detectable performance issue in their lifetime. Under this perspective, supporting developers with methods and techniques for automatically resolving performance issues is a valuable contribution. Despite the relative straightforwardness of manual resolutions of statically-detectable performance issues, the automatic resolution of some of them is far from being technically trivial. For example, the automatic resolution of *HandlerLeak*s involves extensive refactoring, where the resulting code is heavily based on weak

references and object-oriented reachability. Initial steps in this direction are being already performed in the context of Android-specific energy-efficiency optimizations (Cruz and Abreu 2018).

In this context, a relatively high number of apps (124) exhibit the *refactoring* evolution pattern, meaning that at some point of the app's lifetime the number of statically-detectable performance issues has a strong decrease, independently of the amount of changes in the source code. As researchers, we can closely investigate what is happening during the occurrences of a refactoring pattern in order to *learn how developers are actually resolving statically-detectable performance issues*. The results of this analysis can be used as drivers for the design of methods and techniques for (semi-) automatically resolving Android performance issues in the future.

Finally, in the context of this specific study, having an automated refactoring technique will allows us to (i) automatically resolve all the unresolved issues we identified in RQ_3, (ii) submit their resolutions as pull requests in the original GitHub repositories, and (iii) measure the percentage of pull requests that are merged by app developers. All together, those activities will bring a better understanding about how Android developers consider detected and resolved performance issues in real industrial contexts.

PRQ_3 – What is the actual impact of Android performance issues at run-time? Being able to automatically resolve detected issues (see PRQ_2) opens also for the empirical assessment and measurement of the *impact in the resolution of statically-detectable Android performance issues* in terms of, e.g., CPU usage, memory consumption, app's frame rate, etc.

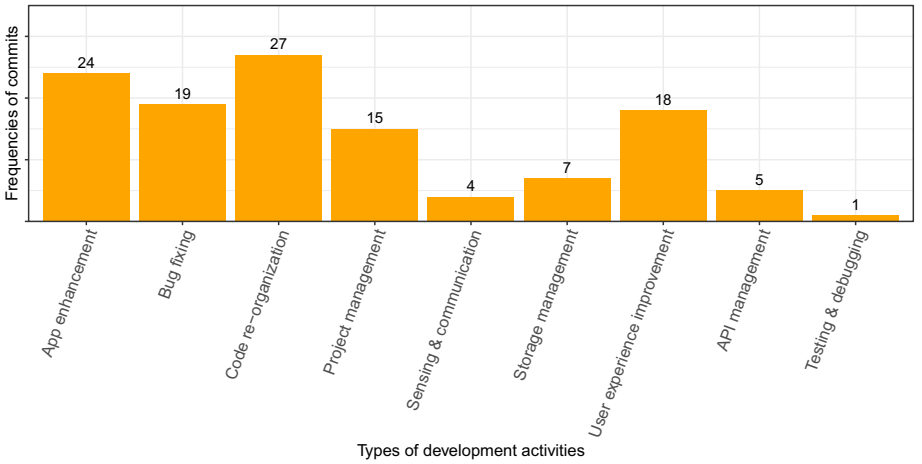
At the time of writing, this line of research has not been explored yet and it can likely lead to an impactful contribution to the body of knowledge in the field of mobile software engineering. Indeed, recently it empirically emerged that a number of Android developers are indifferent to performance issues and they challenge their relevance and impact (Habchi et al. 2018). Clearly, providing empirical evidence about the actual impact of performance issues will help in the overall adoption of static analysis tools like Android Lint, promoting a more careful treatment of performance-related aspects of apps, and thus potentially leading to improving the apps' quality.

PRQ_4 – Are some performance issues more difficult to be resolved? As discussed when answering RQ_3 and RQ_4, some performance issues seem to be more difficult to be resolved, either in terms of code churn or days before the resolution. In order to better understand *why* this phenomenon is happening, we perform a preliminary analysis targeting specific subsets of issue-resolution commits in our dataset. Specifically, we firstly rank all 1,314 commits with resolved issues based on their code churn and then we select the top-10 commits in terms of LOCs for each type of performance issue. This leads to a set of 70 issue-resolving commits (10 for each type of performance issue) with very high code churn (average = 368.7, median = 249.5). At this point, we consider GitHub commit messages and conduct a content analysis session (Lidwell et al. 2010) on all 70 commit messages. Specifically, we categorize them according to the taxonomy of self-reported activities of Android developers proposed and empirically validated by Pascarella et al. (2018). The taxonomy entails a wide variety of different activities at different levels of abstraction (e.g., bug fixes, functionality implementation, release management, access to sensors, etc.). The taxonomy is composed of two levels, where the first layer (9 items) groups together activities with similar overall purpose (e.g., app enhancement, bug fixing, API management), whereas the subcategories (49 items) in the lower level provide a finer-grained categorization (Pascarella

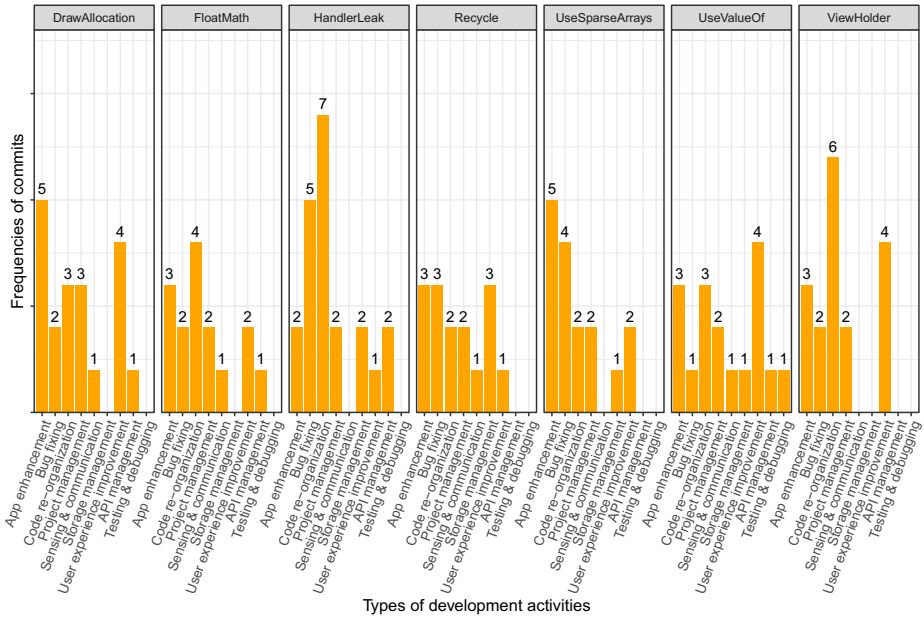
et al. 2018). In this study we focus on the top 9 level-categories only and we assigned one or more categories of development activities to each commit.

Figure 19a and b present how frequently each category of Android developers' activities appears in all **commits with high code churn** in general and across the 7 types of performance-related issues, respectively.

It does not come as a surprise that *code re-organization*, *app enhancement*, *bug fixing*, and *user experience improvement* are the most recurrent types of development activities co-occurring with issue resolutions involving long code churn; indeed, they are also the most



(a) Frequency of development activities in general

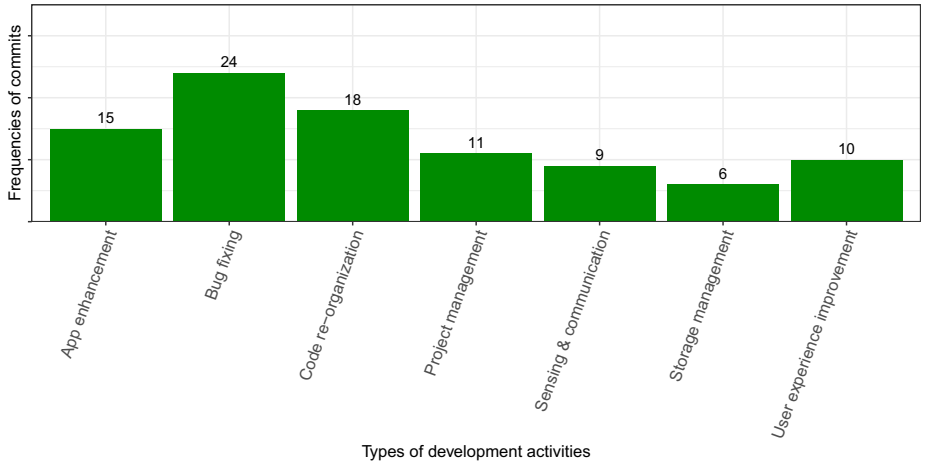


(b) Frequency of development activities across types of performance issues

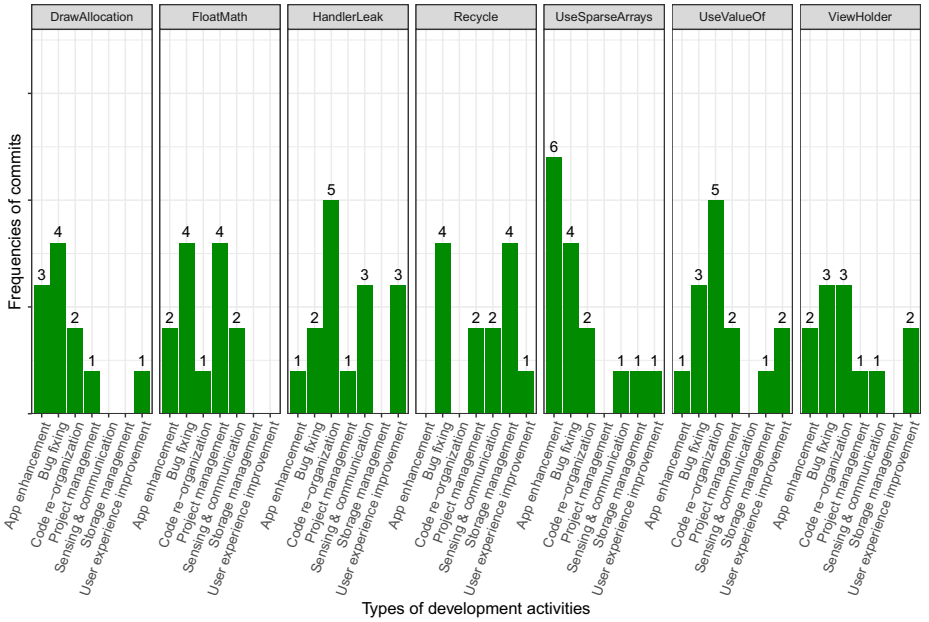
Fig. 19 Development activities performed in issue-resolving commits with high code churn

recurrent activities in the original study in Pascarella et al. (2018), which did not focus on any specific type of commit. Similarly, the most recurrent activity co-occurring with issue resolutions with high code churn is code re-organization (for both $HandlerLeak$ and $ViewHolder$), which includes activities like refactoring, and code cleanup. Also this result is quite expected since resolving those two types of issues can be considered as special cases of code refactoring or cleanup.

We consider also the cases requiring **low code churn** and analyze the resulting data with the same procedure we applied before and focusing on the issue-resolving commits with



(a) Frequency of development activities in general



(b) Frequency of development activities across types of performance issues

Fig. 20 Development activities performed in issue-resolving commits with low code churn

the lowest code churn. The resulting subset is composed of 70 commits (10 for each type of performance-related issue) and has an average of 22.61 and a median of 10.50 LOCs per commit. As shown in Fig. 20a and b, also when considering commits with low code churn we did not get patterns extremely different from what we observed in commits with high code churn. Overall, in many cases we could observe that the resolution of performance issues occurs together with other potentially unrelated development activities (e.g., app enhancement or bug fixing). This may be an indication that Android Lint is commonly used as part of everyday development activities, potentially thanks to its default integration to the Android Studio IDE. This finding is also confirmed in Habchi et al. (2018), where many Android developers reported that they prefer to (i) use Android Lint from the project startup and (ii) try to keep the code as clean as possible by frequently considering Android Lint in their development workflow.

We performed the analyses described above in order to get an initial indication about what developers are doing contextually to the resolution of performance issues. In many cases we could observe that the resolution of performance issues occurs together with other potentially unrelated development activities (e.g., app enhancement or bug fixing). This may be an indication that Android Lint is commonly used as part of everyday development activities, potentially thanks to its default integration to the Android Studio IDE. This finding is also confirmed in Habchi et al. (2018), where many Android developers reported that they prefer to (i) use Android Lint from the project startup and (ii) try to keep the code clean by frequently considering Android Lint checks. With only 280 data points, we are aware that the performed analyses have low statistical power. Nevertheless, a more in-depth analysis about the root causes of low-high code churns and number of days before resolution is surely a worthwhile future research direction. For the interested researchers, the raw data we manually collected so far is available in the replication package of this study.

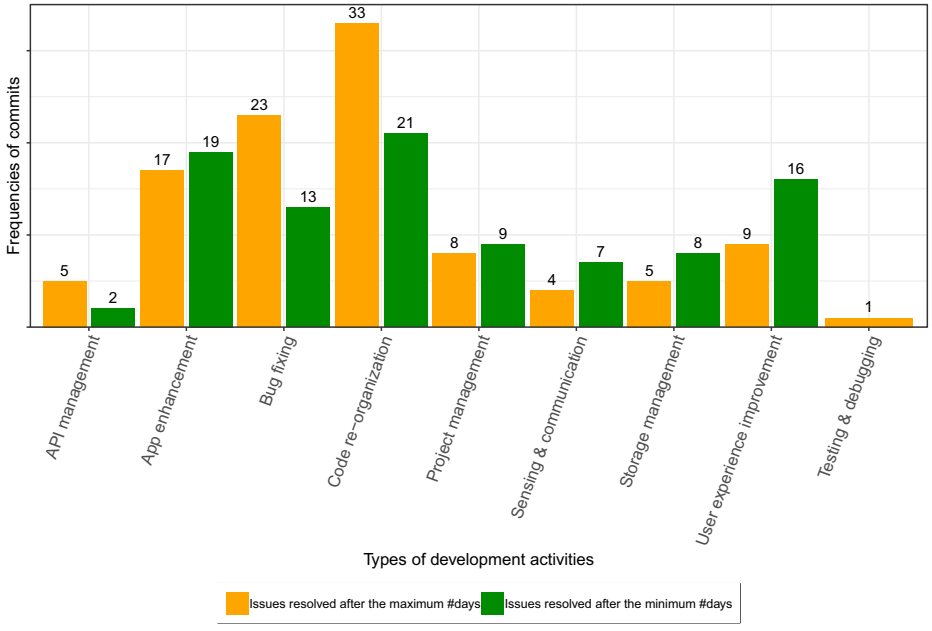


Fig. 21 Development activities performed in issue-resolving commits with min and max resolution time

We also noted that some issues have a very short resolution time (even less than one day in some cases). We performed an additional analysis similar to the one about code churn with a focus on the top-10 commits in terms of issue resolution time for each type of performance issue. Figure 21 presents those developers' activities co-occurring with issue-resolution commits with extremely low (average = 2.62, median = 0.22) or high (average = 582,568, median = 236) resolution times in days. Similarly to what we observe also for code churn, the most recurring development activities tend to co-occur with issues with both extremely high and low resolution times.

Finally, only in about 10% of issue resolution commits developers explicitly mention that they resolved a performance issue. This phenomenon can be seen as an indication of the fact that *the resolution of statically-detectable performance issues is embedded into other activities when developing Android apps*. One explanation of this phenomenon may be the integration of Android Lint into Android Studio, where the results of the analysis are directly integrated into the development environment, without context switches, tool configuration, etc. The same trend has been confirmed also in a recent industrial study at Google, where the results of static analyses are taken more into consideration when they are integrated into the development workflow, executed at compile time, and enabled by default for everyone (Sadowski et al. 2018).

10 Threats to Validity

This section discusses the main threats to the validity of our study and the countermeasures we applied for mitigating them.

Construct Validity threats are related to the relationship between theoretical knowledge and actual observations.

We detected performance issues by using only one static analysis tool, i.e., *Android Lint*. This decision involves the risk of having a mono-method bias in our study (Wohlin et al. 2012) since the results of the whole study are based exclusively on the types of issues supported by the current implementation of *Android Lint*. This means that other types of performance issues may be present in the analyzed apps, but are not considered in our study because they may not be supported by *Android Lint*. As discussed in Section 2, there is a number of other static analysis tools which are applicable to Android apps, such as *FindBugs*, *PMD*, *PerfChecker*, *Paprika*. Those tools could have been used in combination with *Android Lint*, in order to complement and cross-check its results and have a better coverage of statically-detectable Android performance issues. However, at the time of writing, *Android Lint* is the only static analysis tool which (i) is dedicated to Android-specific issues at the source code level and (ii) has a specific category for performance-related issues. We decided to use only *Android Lint* in order to do not confound the results of the study by considering issues outside the scope of the experiment (i.e., statically-detectable performance issues in Android apps), such as maintainability smells, bugs, security vulnerabilities, etc. Moreover, in this study we put ourselves in the same conditions as the developers, who generally use Android Lint because it is integrated and activated by default in Android Studio. As confirmed also by other studies, having a linter integrated in the common development workflow makes it more trustable by developers (Habchi et al. 2018; Sadowski et al. 2018). This makes us reasonably confident about the representativeness of the results of our study, especially when dealing with the resolution and lifetime of the performance issues identified by Android Lint. We do not have empirical evidence about the adoption in practice of linters developed in academia, e.g., *Paprika*.

Nevertheless, the usage of Android Lint allows us to cover a relatively large set of performance-related issues, ranging from low-level issues (e.g., *UseValueOf*) to more encompassing ones (e.g., *HandlerLeak*).

When executing the experiment, Android Lint was supporting 9 performance-related checks, whereas its current latest version supports 36 performance checks (as of June 2019). In order to better understand which Android Lint checks could have been included in our study by considering the latest version of Android Lint,

we analyzed the current implementation and documentation of Android Lint and found out that we could support 4 additional performance checks, namely: `WakelockTimeout`, `StaticFieldLeak`, `LogConditional`, and `SyntheticAccessor`. However, at the time of the computation this was not considered and will be a matter of future investigation. As a way to mitigate this potential threat to validity, we make the data about all 36 performance-related checks in Android Lint publicly available in the replication package of the study.

As many other static analysis tools, *Android Lint* can suffer from the presence of false negatives, i.e., performance issue not detected. Nevertheless, the number of performance issues detected by *Android Lint* is relatively large, making us reasonably confident about the considerations we give when analyzing their evolution over time. Indeed, it is out of the scope of our study to precisely identify *all* performance issues of Android apps, whereas our main objective is to characterize how statically-detectable ones evolve over the lifetime of Android apps, how much time they remain in the code base, and how developers actually resolve them. Moreover, our study may potentially suffer from the presence of false negatives, i.e., performance issues actually present in the app but are not detected by Android Lint. This potential bias is mainly due to (i) the fact that we rely on the heuristics and checks implemented in *Android Lint* and (ii) that we exclusively rely on *Android Lint* for the detection of performance issues. As previously discussed, we decided to focus exclusively on *Android Lint* in order to keep the experiment as focused and realistic as possible (in terms of developers' conditions) and because *Android Lint* currently is the only static analysis tool which has a dedicated category of checks related to performance and it is specific for Android. As future work, as more static analysis tools for Android performance issues will possibly emerge, we will replicate this study and complement our results with those obtained from other tools. As a way to complement the findings emerging from this study, in future work we will assess the actual impact of the detected issues by dynamically analyzing the considered apps. This can be useful to gain evidence-based insights about the practical consequences of the performance issues detected by Android Lint.

It is important that the toolchain for extracting the performance issues across all commits is implemented and configured correctly. We mitigated this potential threat to validity by carefully designing the whole toolchain (see Section 3.3), by testing each component of the toolchain in isolation via repositories for which we knew already the expected outcomes of each data extraction step, and by making the implementation of the toolchain publicly available for independent verification and replication (see the replication package of this study).

Another threat is related to the degree to which the selected apps are representative of the target population (i.e., Android apps published in the Google Play store). We mitigated this threat by considering a relatively large initial set of Android apps (4,287) and by performing an in-depth data quality assurance and filtering process (see Section 3.2).

When considering issue resolutions, there might have been cases of "accidental" performance issue resolutions, when a source code fragment was deleted for other reasons. Nevertheless, those cases are not jeopardizing the results of the study since our aim is to establish the lifetime of performance issues, independently of whether the resolution of

the issues is conscious or not. In order to better characterize to what extent developers are consciously resolving performance issues, in RQ_4 we report on the documented resolutions of issues only. Also, for RQ_1, RQ_2, and RQ_3 we partially mitigated this potential source of bias through a line-based tracking of the issues over time via the LHDiff tool (see Section 3.3).

When answering RQ_3, we consider the CDFs globally with respect to each type of performance issue, instead of considering them on a per-app basis. This may be a threat to validity since the lifetime of performance issues may vary depending on the potentially different maintenance strategies of each individual app. In this context, app-specific factors may influence the obtained distributions, potentially missing the opportunity to make a more fine-grained analysis. However, we decided to analyze the CDFs globally with respect to each type of performance issue due to the relatively limited number of data points we could have obtained when considering each app in isolation.

When answering RQ_4, we are assuming that if a commit message contains specific keywords, it is describing the resolution of a performance issue. We are aware that such an approach may miss commits where the resolution of the performance issue is not documented. False positives have been avoided by performing a manual analysis of all identified issue-resolution commits.

Conclusion Validity is about the relationship between treatment and outcomes of the study.

We carefully took into consideration the assumptions of each applied statistical test. We minimized the possibility of misleading results by relying on non-parametric tests, such as the KS test.

The qualitative analysis we performed when answering RQ_1 is based on the manual categorization of issue evolution plots, potentially leading to the subjective interpretation of evolution patterns. We mitigated this potential threat to validity by (i) carefully following the open card sorting methodology (Spencer 2009), (ii) involving three researchers, who worked both independently and collaboratively across the various analysis phases, and (iii) statistically assessing the level of agreement between the involved researchers via the Cohen-Kappa statistics (Cohen 1968).

We mitigated the above mentioned potential threats to validity by preparing a full replication package of the study containing the raw data and statistical data analysis scripts, thus making the data analysis phase of this study fully reproducible.

Finally, in the preliminary analysis presented in Section 9.3 (implication R4), we use both code churn and issue resolution time as proxies of the difficulty of resolving issues. However, as it also emerged from that preliminary analysis, massive code churn and long issue resolution times may be due to developers performing other activities that are not related to performance issues (e.g., resolving bugs or implementing new features). As a future work we will mitigate this potential threat to validity by (i) identifying the subset of commits in which developers are working exclusively on the resolution of performance-related issues and (ii) carrying out a more in-depth analysis only on those commits commits.

Internal Validity is related to factors internal to our study that can influence our results.

During the dataset building phase, we noticed that many potentially-relevant GitHub repositories were actually not containing apps (e.g., repositories hosting only Android libraries). As discussed in Section 3.2, we discarded those types of repositories from our dataset. Moreover, despite all considered repositories are about the implementation of Android apps, their structure can heavily differ in terms of folders and files organization. This means that it is possible to obtain false results by considering non-app related source

code in the static analysis (e.g., third-party libraries, code implementing the back-end of the apps, code developed for other platforms). We mitigated this potential threat to validity by identifying, for each repository, the app's root folder containing its source code. Then, the execution of Android Lint has been set up so to consider only the source code contained within the app's root folder.

External Validity is related to the generalizability of the obtained findings. Due to our requirement of having access to the full versioning history of the apps, this study considers exclusively Android apps whose source code is available in GitHub, which may be not representative of the population of all Android apps. However, as we study how apps evolve over time, we need access to the previous versions of the app. Mining GitHub grants access to fine-grained snapshots of each app, whereas in Google Play developers publish only the official releases of their apps. Moreover, we are interested in how performance issues are introduced and resolved by developers in the Java code of their apps; in Google Play only the binary code of the app is available, which may be structurally different from the source code produced by developers e.g., because of code obfuscation. Nevertheless, the built dataset has a high heterogeneity, both in terms of apps size, number of contributors, lifetime, and categories. Moreover, in order to further mitigate this potential threat to validity, we ensured that all considered apps are also distributed in the Google Play Store, meaning that they are real apps being actually used, and not on-time demos or toy examples.

11 Related Work

Existing research related to our study has been mainly carried out in the context of (i) empirical studies on Android apps performance (Section 11.1) and (ii) the evolution of statically-detectable bugs and issues (Section 11.2).

11.1 Empirical Studies on Android Apps Performance

Liu et al. (2014) conducted an empirical study involving 70 occurrences of performance bugs from 8 real-world Android apps. The study identified three common types of performance bug, i.e., GUI lagging, energy leak, and memory bloat. GUI lagging resulted to be the more frequent performance bug (53/70), followed by energy leak (10/70), and memory bloat (8/70). Furthermore, the authors of this study developed PerfChecker, a tool for detecting two types of performance bugs: (i) violations of the view holder pattern and (ii) lengthy operations in the UI thread. Also, the study revealed that debugging and resolving performance bugs are generally more difficult than debugging and resolving non-performance bugs, confirming the need to further investigate on Android performance (either via static analysis or via run-time profiling). Differently, from the study proposed by Liu et al., our study involves a much larger number of apps (724) and we are reusing *Android Lint* to investigate nine types of performance issues, instead of developing our own analysis toll. Moreover, we also analyze the lifetime of performance issues throughout the whole duration of the projects in GitHub, which gives us a deeper understanding about the severity of the considered types of performance issue, e.g., which kinds of performance issues tend to remain in the code base and which kinds of issues take more time to be resolved.

Vásquez et al. (2015) interviewed 485 developers of open-source Android apps and libraries about their best practices for tackling performance issues. The results of the study revealed that Android developers (i) rely on multi-threading to prevent/avoid long operations in the main thread, (ii) perform GUI optimization to reduce the complexity of the UI

of their apps, (ii) cache results in their apps in order to improve the time to access resources, and (iii) focus on good memory management to avoid heavy executions of the garbage collector and memory leaks. We can consider our study as complementary to the one by Linares-Vasquez et al. Indeed, the main goal of our study is to characterize the presence and evolution of statically-detectable performance issues, rather than analyzing the best practices applied by developers when fixing performance issues. This fundamental difference of our objectives is reflected also in the design of the studies, where we focus on mining software repositories techniques for inspecting the versioning history of the considered apps, as opposed to an online survey involving practitioners.

Nistor and Ravindranath (2014) proposed a technique called SunCat for allowing developers to use common small inputs to understand potential performance problems that apps could have for larger inputs.

They evaluated their proposed technique by considering 29 different scenarios of 5 Windows phone apps, which unveiled the presence of 9 performance problems. Our study differs from their, since we aim at empirically characterizing the evolution of performance issues along the lifetime of Android apps, instead of focusing on a new technique for performance bug identification.

Gomez et al. introduced DUNE, a context-aware approach for identifying UI performance regressions among different Android app releases and heterogeneous contexts (Gómez et al. 2016). DUNE works in two steps: it firstly constructs a model of the UI performance metrics coming from the execution of a test suite targeting the app, then it flags potential UI performance deviations in new test runs. In this context, DUNE is able to identify the specific UI events that may potentially trigger a UI performance issue and to characterize the context in which it occurred (e.g., a specific Android SDK version). DUNE has been empirically evaluated on 3 Android apps. In our study, we reuse Android Lint for our analysis along the lifetime of a large set of real Android apps; we did not use the DUNE tool as it needs to run and measure the apps at run-time, making the execution of the whole experiment prohibitively long with respect to our available resources.

Furthermore, a number of studies have been conducted about the use of profiling techniques to measure mobile apps for supporting their performance optimization (Qian et al. 2011; Ravindranath et al. 2012). For instance, the authors of Qian et al. (2011) monitored the interaction between the resource management layer and the application layer to identify inefficiencies in the usage of those resources that are mostly responsible for poor performance in mobile apps. Our study is different from the profiling-based ones as we build on static analysis tools and techniques for performing our analysis, instead of focusing on run-time profiling. Moreover, in this study, we are mainly focusing on how performance issues evolve over the whole lifetime of the code base of the mobile app, as opposed to investigating on the performance of the app at a given release.

Habchi et al. interviewed 14 Android developers to investigate on the motivations, practices, and constraints in using Android Lint for performance purposes (Huchard et al. 2018). Their study is observational in nature and they followed a qualitative research approach based on Grounded Theory concepts. Among the various findings, this study revealed that Android Lint can benefit developers to learn about the Android framework, anticipate performance bottlenecks, and easily identify performance bad practices. Moreover, the study provided insights on how Android Lint is used, e.g., in some cases Android Lint was required on a team level, but also for supporting individual development, and for prioritizing performance aspects of the Android app. Finally, the study highlighted also some obstacles against the adoption of Android Lint, such as the fact that performance is managed only reactively in some organization, the perception that linters are not suitable for performance

analysis, and the fact that analysis results are not well presented to developers. Differently from the study by Habchi et al., in this work we are having an orthogonal perspective and we aim at characterizing the evolution of performance issues supported by Android Lint over time. Also, in this study we are investigating on the lifetime of performance issues identified by Android Lint and provide insights about how developers are resolving them in open-source Android apps.

Cruz et al. conducted a study on six Android apps with the aim of analyzing whether performance-related issues identified by Android Lint can have also an impact on energy consumption (Cruz and Abreu 2017). The most relevant finding of this study is that resolving a subset of the performance issues detected by Android Lint can save up to one hour of battery life of the mobile device. The performance issues correlated with the battery savings are the following: *ViewHolder*, *DrawAllocation*, *WakeLock*, *ObsoleteLayoutParam*, and *Recycle*. Moreover, the authors of Cruz and Abreu (2017) propose an approach that takes as input the source code of an Android app and automatically refactors the energy-greedy issues mentioned above. Differently, in our study we focus on the time dimension and on how performance-related issues evolve over time, which ones remain in the code base longer, and how they are resolved by developers in the context of real open-source projects. Finally, the research questions guiding our study are different, and complementary, with respect to the study by Cruz et al.. Specifically, we aim at characterizing the presence and resolution time of performance-related issues in Android apps), whereas Cruz et al. aim at providing empirical evidence about the relationship between statically-detectable performance issues in Android apps and their energy consumption. This difference profoundly impacts the design and the number of subjects involved in the two studies, i.e., a longitudinal study involving 724 apps vs a small-scale measurement-based empirical study involving 6 apps.

11.2 Empirical Studies on the Evolution of Statically-Detectable Issues

In the previous section, we discussed the *Paprika* tool. In Hecht et al. (2015a) the same tool has been used for assessing the evolution of quality metrics of Android apps over time. Here, PAPRIKA has been configured to consider 3 Object-Oriented antipatterns and 4 Android-specific antipatterns (namely, Member Ignoring Method, Leaking Inner Class, and UI Overdraw), and the detected antipatterns are utilized as proxies for the quality of the app. *Paprika* has been executed on multiple releases of 106 Android apps, which have been collected from the Google Play Store. Then, (i) the baseline of software quality has been computed from the whole set of collected apps and (ii) the quality of each release of each app has been estimated as the deviation from the baseline. The study revealed the presence of relationships between different antipatterns (e.g., the blob and complex class antipatterns tend to evolve together) and established 5 major quality evolution trends, similar to the ones we identified in Section 5. Our study differs from Hecht et al. (2015a) since we focus exclusively on performance-related issues and go deep into both the context in which performance issues are introduced and resolved in the source code of the app. This has been possible thanks to our dataset building strategy, which allowed us to have a much finer level of granularity (i.e., at the single commits level in GitHub, which may occur in the order of minutes), as opposed to focusing only on the official app releases in the Google Play Store, which may have a timespan of even several weeks. Finally, we have a dedicated analysis of the lifetime of each type of performance issue and discuss the documented resolutions of performance issues.

Di Penta et al. conducted an empirical study on the evolution of statically-detectable vulnerabilities at the source code level (Di Penta et al. 2009). Vulnerabilities have been

detected by using three different tools, i.e., Splint, Rats and Pixy. The objects of the study were 3 highly-popular open-source networking applications (i.e., Squid, Samba and Horde). Similar to what we did with *Android Lint*, Di Penta et al. (2009) analyzed the evolution of vulnerabilities by running analysis tools over the whole lifetime of the 3 projects. The study reported that most of the vulnerabilities tend to be resolved from the system. Our study differs from the one by Di Penta et al. primarily in the goal and subjects (i.e., we are interested in performance issues of Android apps, whereas they are interested in the security aspects of generic software. Methodologically, the two studies are similar since both of them (i) focus on the number and evolution patterns of statically-detectable issues and (ii) assess whether the CDF of each type of issue can be modeled using known statistical distributions. However, in our study we consider a much larger dataset (724 projects against 3) and our collected data has a finer level of granularity since we executed *Android Lint* on *all* commits of all 724 apps, whereas Di Penta et al. (2009) used a time-windowing approach.

Chatzigeorgiou and Manakos (2014), analyzed the evolution of four bad code smells (i.e., Long Methods, Feature Envy problems, State Checking smells and God class) throughout the lifetime of two large open-source applications (i.e., JFlex and JFreeChart). The main finding of this study is that bad code smells tend to persist up to the latest versions of the project. Moreover, they performed a survival analysis on the collected data and discovered that once the smell is introduced, it tends to remain for a long time in the code base. Also, their results indicated that few smells are resolved from the system, and most of their resolutions are not due to refactoring activities, but rather are unintentional (e.g.,, they happen when changing other parts of the source code).

Tufano et al. performed a survival analysis in their large-scale study on bad code smells (Tufano et al. 2017). Their study involved the analysis of each commit of 200 open-source systems mined from GitHub. Their study showed that 80% of the investigated code smells survive in the system and a very small percentage (9%) is resolved due to some particular refactoring operations. Our study mainly differs from the studies of Chatzigeorgiou and Manakos (2014) and Tufano et al. (2017), because of its focus on statically-detected performance issues of Android apps, as opposed to bad code smells in open-source projects.

12 Conclusions and Future Work

Mobile applications (apps) are nowadays becoming more and more rich of complex features, and are continuously updated over time to cope with users' requests. Poor design and implementation choices upon evolving apps can lead towards performance problems. While it is often the case that developers identify performance tools through profiling and testing tools, an appropriate usage of static analysis tools could constitute a cheaper complement to that.

In this paper we investigate how performance problems — detected by a static analysis tools, i.e., Android Lint[25] — occur, evolve and eventually disappear in Android apps. More specifically, we analyze a set of 724 popular open source Android apps, and we found performance issues in 316 of them.

Results of the study indicate that:

1. Issues due to the lack of recycling data collections and other resources such as database cursors are the most frequently occurring ones, while Android Lint found very few instances of *ViewTag* and *WakeLock* issues.

[25] Android studio project site. http://tools.android.com/tips/lint.

2. Performance issues in general tend to appear suddenly in a file rather than being gradually introduced, and remain in the system for a long time (and number of commits).
3. Some issues, primarily related to the user interface and to memory management, are either resolved (in our observation period) or tend to remain in the app for a longer time, while other issues (primarily of algorithmic nature) tend to be resolved quickly (when they are resolved), possibly because there are well-known, easy solutions for them.
4. We found performance issue resolutions to be documented in commit messages in 10% of the cases. This may either indicate that in other cases they were (accidentally) resolved along with other changes, or that in any case their resolution was not considered as the primary goal of the commit.

Future work will aim to replicate the study by considering other static analysis tools and a larger set of mobile apps with a focus on assessing the differences between the evolution of issues identified by Android Lint and those identified by other static analysis tools. Also, based on the analysis of issue resolution patterns, we are developing a recommender system aimed at suggesting solution patterns for certain kinds of performance issues. Moreover, we will perform an in-depth qualitative study on the identified evolution patterns in order to better understand the context and main motivations behind them, so to better support developers and researchers in understanding how to prevent the occurrence of negative patterns such as issue-rich features and sticky issues. A deep investigation on the solutions implemented by Android developers will be extremely valuable in order to better understand the most convenient resolutions in terms of e.g., effort, change impact, code understandability; in this context a tool for automatically extracting the characteristics of the implemented resolutions will be needed, together with a proper validation of its accuracy (e.g., in terms of precision and recall). Finally, the analysis of the resolution patterns can potentially lead to the development of a tool for automatically resolving statically-detectable performance issues, which will open for the possibility of getting better insights about the actual impact of issues on the overall performance of the app (Habchi et al. 2018) and about how developers perceive the resolutions proposed by the tool.

Appendix A: Examples of Android Performance-Related Issues

```
protected void presetClicked(int i) {
Log.i(TAG, "presetClicked "+i);
TypedArray preset_vals = mContext.getResources().obtainTypedArray(
mContext.getResources().getIdentifier(
"presets"+mFnId+"_"+i, "array", mContext.getPackageName()));
for(int j=0; j<mSliders.size(); j++) {
float val = preset_vals.getFloat(j, 0);
Log.i(TAG, "slider["+j+"]="+val);
mSliders.get(j).setValue(val);
}
}
```

Listing 2 Example of `Recycle` issue (`dstahlke/rdn-wallpaper` - src/org/stahlke/rdnwallpaper/PresetsBox.java)

```
//The Special Key Codes
mSpecialCodes = new Hashtable<Integer, String>();
mSpecialCodes.put(new Integer(-900), "++");
mSpecialCodes.put(new Integer(-901), "--");
mSpecialCodes.put(new Integer(-902), "&&");
mSpecialCodes.put(new Integer(-903), "||");
// ...
mSpecialCodes.put(new Integer(-906), "==");
mSpecialCodes.put(new Integer(-909), "!=");
```

Listing 3 Example of bursty occurrences of the `UseValueOf` issue (`dyne/ZShaolin` - termapk/src/com/spartacusrex/spartacuside/keyboard/TerminalKeyboard.java)

```
private void randomizeOptions(){
HashMap<Integer, Word> optionsMap = new HashMap<Integer, Word>();
for(int i = 0 ; i < optionsList.size() ; i++){
while(true){
int rand = (int)((Math.random() * 10) % 5);
if(optionsMap.containsKey(rand)){
continue;
}else{
optionsMap.put(rand, optionsList.get(i));
break;
}
}
}
```

Listing 4 Example of `UseSparseArrays` issue (`ric03uec/cramit` - src/com/dev/cramit/models/Problem.java)

```
// TODO make this Handler static to prevent memory leaks
private Handler communicatorServiceHandler = new Handler() {
/**
* Needs to know which parameters are passed back in which predefined
* fields of the {@link Message}. </p>
* <ul>
* <li><code>what</code> - hash of the related
* {@link IntentAction.WebService} constant</li>
* <li><code>arg1</code> - one of the constants declared in
* {@link WSConstants.Result}</li>
* <li><code>obj</code> - the returned {@link Bitmap}</li>
* </ul>
*/
```

Listing 5 Example of `HandlerLeak` issue (`mobiRic/StackFlairWidget` - src/com/mobiric/ stackflairwidget/service/ FlairWidgetService.java)

```
private final Handler mHandler = new Handler() {
@Override
public void handleMessage(Message msg) {
switch (msg.what) {
case MESSAGE_UPDATE_HARDWARE_PARAMETERS:
updatedHardwareSettingsValues();
break;
}
}
};
```

Listing 6 Example of `HandlerLeak` issue (`alistairdickie/ BlueFlyVario _Android` - src/com/bfv/hardware/ HardwareListActivity.java)

```
protected void onDraw(Canvas canvas) {
List<Double> numbers = new ArrayList<Double>();

Employee employee = new Employee(_hoursWorked, _this);
//numbers.add(Double.parseDouble(employee.gross()));
numbers.add(employee.payeDouble());
numbers.add(employee.studentLoanDouble());
numbers.add(employee.kiwiSaverDouble());
numbers.add(employee.nettDouble());
double total = employee.grossDouble();

List<Integer> colours = new ArrayList<Integer>();
colours.add(Color.BLUE);
colours.add(Color.GREEN);
colours.add(Color.RED);
colours.add(Color.MAGENTA);
colours.add(Color.YELLOW);
Pie p = new Pie(500);
Paint wallpaint;
```

Listing 7 Example of `DrawAllocation` issue (`kurtmc/MyEarnings` - src/com/mcalpine development/calculatepay/ CalculateActivity.java)

```
// if values before left of decimal are the same -> need to show float decimals
if (android.util.FloatMath.floor(min) ==   android.util.FloatMath.floor(max))
{
minS = Float.toString(min);
maxS = Float.toString(max);

// show same length decimals!
if (minS.length() > maxS.length())
{
minY.setText(minS.substring(0, maxS.length()));
maxY.setText(maxS);
}
else
{
minY.setText(minS);
maxY.setText(maxS.substring(0, minS.length()));
}
}
else   // otherwise only show integers
{
minY.setText(Integer.toString((int)(min)));
maxY.setText(Integer.toString((int)(max)));
}
```

Listing 8 Example of `FloatMath` issue (`dirktrossen/AIRS` - src/com/airs/TimelineActivity.java)

```
@Override
public View getView(int position, View view, ViewGroup parent) {
LayoutInflater inflater = (LayoutInflater)
     getContext().getSystemService(Context.LAYOUT_INFLATER_SERVICE);
view = inflater.inflate(R.layout.text_with_delete, parent, false);

final String textItem = getItem(position);
TextView textView = (TextView) view.findViewById(android.R.id.text1);

textView.setText(textItem);
// add listener to the delete button
Button button = (Button) view.findViewById(android.R.id.button1);
button.setOnClickListener(new OnClickListener() {
@Override
public void onClick(View v) {
//delete button clicked
onDeleteListener.onDelete(textItem);
}
});

return view;
}
```

Listing 9 Example of `ViewHolder` issue (`asksven/BetterWifiOnOff` - BetterWifiOnOff/src/com/asksven/betterwifionoff/CreditsAdapter.java)

Appendix B: Catalog of Solutions for Statically-Detectable Performance Issues

In the following, we present a catalog of solutions for statically-detectable performance issues as emerged from our analysis. All the examples of solutions are available in Appendix B.

Recycle To resolve the `Recycle` issues in the project *uberspot/AnagramSolver*, developers purposely add the lines of code to close the cursor properly after being used. It can be seen in code snippet (Fig. 22 in Appendix B) that developers use *cursor.close()* at the end (at line 174) to close the database query cursor properly and this changes in source code led to resolution of `Recycle` issues from the project. Whereas, in other (Recycle) category, `Recycle` issue was solved in the project *shlusiak/Freebloks-Android* by recycling the resource, i.e., by calling the *p.recycle()* method (see lines 540 and 569 in Fig. 23).

ViewHolder To resolve this issue developers should implement `ViewHolder` pattern in *getView()* callbacks. For example, it can be noted from the manual observation that developers specially implemented `ViewHolder` Pattern for ExerciseImageListAdapter to resolve the `ViewHolder` issue from the project chaosbastler/opentraining (an example from our dataset) as shown in Fig. 24. The idea is to reuse earlier recycled list items. It prevents the inflation of list items layout when there are recycled items available for reuse (Lines 99-100). When the list of items is created for the first time, the references to inner view object

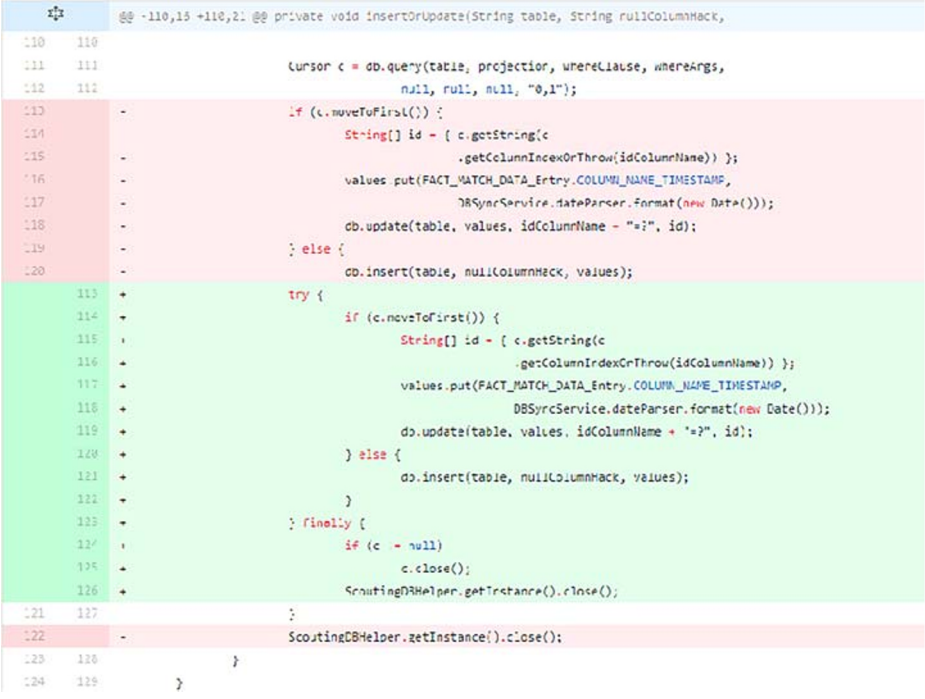


Fig. 22 Example of resolving of the *Recycle* issue (`username115/FRCScouting` - src/org/frc836/database/DB.java)

Fig. 23 Example of resolving of the *Recycle* issue (`shlusiak/Freebloks-Android` - src/de/saschahlusiak/freebloks/game/FreebloksActivity.java)

are identified and stored in a particular data structure for reuse (Lines 95-96). The main advantages of using the `ViewHolder` pattern are that (1) it can save computation for inflation of list items layout by reducing *findViewById()* calls and also invokes less inner view retrieval computations, and (2) it is memory efficient for building new list items. Furthermore, to reduce the impact of frequently invoked callbacks in *getView()* implementation, developers should use this kind of efficient design.

HandlerLeak To resolve the `HandlerLeak` type of issues in the project *stdev293/ battery-waster-android*, developers declared the handler as static class. It can be clearly shown from Fig. 25, that developers create a new static class for the handler to avoid the potential memory leaks. Moreover, developers also used a `WeakReference` to outer class and pass the object to instantiate a handler.

```
-   public View getView(final int position, View convertView, ViewGroup parent) {
-       //View vi = convertView;
-       View vi = mInflater.inflate(de.skubware.opentraining.R.layout.exercise_image_list_row, null);
-
-       ImageView imageView = (ImageView) vi.findViewById(R.id.exercise_image);
-       imageView.setImageBitmap(getBitmap(position));
-       imageView.setOnClickListener(new OnClickListener(){
-           @Override
-           public void onClick(View v) {
-               ShowImageDialog.showImageDialog(getBitmap(position), mContext);
-           }
-       });
-
-
-       return vi;
+       ViewHolderItem viewHolder;
+
+       if(convertView==null){
+       // inflate the layout
+       convertView = mInflater.inflate(de.skubware.opentraining.R.layout.exercise_image_list_row, null);
+
+
+       // set up the ViewHolder
+       viewHolder = new ViewHolderItem();
+       viewHolder.imageViewItem = (ImageView) convertView.findViewById(R.id.exercise_image);
+
+       // store the holder with the view.
+       convertView.setTag(viewHolder);
+
+   }else{
+       // avoided calling findViewById() on resource each time, use the viewHolder
+       viewHolder = (ViewHolderItem) convertView.getTag();
+   }
```

Fig. 24 Example of resolving of the `ViewHolder` issue (`chaosbastler/opentraining` - app/src/de/skubware/opentraining/activity/create_exercise/ExerciseImageListAdapter.java)

UseSparseArrays We manually analyzed the source code of several projects to know how the `UseSparseArrays` issue type is resolved, we observed that for better performance of memory server storage, developer removed HashMaps and used `UseSparseArrays` instead. One of the sample documented example can be shown in the Fig. 26(from our dataset) where the `UseSparseArrays` type of issue is resolved from the project `pocmo/Yaaic`.

However, `UseSparseArrays` are assumed to be more memory efficient and trigger less garbage collection as compared to its counterpart `HashMaps` with no key impact on operations performance of maps. Moreover, `SparseArrays` allocate less memory as compared to `HashMaps`.[26]

DrawAllocation Related to `DrawAllocation` issue category, we consider the *mchow01/FingerDoodle* project to analyze the solution.

From our manual analysis, we observed that developer removed the Paint object from *onDraw()* function (i.e., a background Paint object is created each time when the draw operation takes place and memory is allocate every time) as shown in Fig. 27 (Appendix B). To resolve this issue, developer pre-allocate the background Paint (at line 18) upfront (i.e., outside from *onDraw()* function) and reuse it instead, which will prevent from the UI lag since memory will not allocate at each time when *onDraw()* or *Layout()*function is called. Therefore, it is a good suggestion for developers to allocate new object before the draw or layout operation.[27]

[26] Sparsearray vs hashmap. http://www.sable.mcgill.ca/soot/.

[27] Android lint checks.

```
+ import java.lang.ref.WeakReference;
+
+ import android.os.Handler;
+ import android.os.Looper;
+ import android.os.Message;
+ import android.view.View;
+
+ import com.stdev293.batterywasterdemo.activities.BatteryWasterActivity;
+
+ public class SinksControlThread extends Thread {
+       // ------------------------------------------------------------------------------------------
+     // Constants
+       // ------------------------------------------------------------------------------------------
+       public static final String THREAD_NAME="SinksControl";
+       public static final int ACTION_START_ALL = 0;
+       public static final int ACTION_STOP_ALL = 1;
+       public static final int ACTION_LIGHT_ON = 2;
+       public static final int ACTION_LIGHT_OFF = 3;
+
+       // ------------------------------------------------------------------------------------------
+     // Members
+       // ------------------------------------------------------------------------------------------
+       private static SinksControlActionHandler handler = null;
+
+       private BatteryWasterActivity mActivity;
+
+       // ------------------------------------------------------------------------------------------
+     // static Handler class to carry out actions in the thread
+       // ------------------------------------------------------------------------------------------
+       private static class SinksControlActionHandler extends Handler {
+               private WeakReference<BatteryWasterActivity> mActivityRef;
```

Fig. 25 Example of resolving of the `HandlerLeak` issue (`stdev293/battery-waster-android` - src/com/stdev293/batterywasterdemo/sinks/SinksControlThread.java)

FloatMath For the `FloatMath` issue type, instead of `FloatMath` declaration, developers should use `Math` in the source code to resolve this issue. It can be noted from the Fig. 28 (Appendix B), that after depreciated from `FloatMath` to `Math` (as developer wrote in

```
35  35    import android.database.DatabaseUtils;
36  36    import android.database.sqlite.SQLiteDatabase;
37  37    import android.database.sqlite.SQLiteOpenHelper;
    38  + import android.util.SparseArray;
38  39
39  40    /**
40  41     * Database Helper for the servers and channels tables
    @@ -333,9 +334,9 @@ public void setCommands(int serverId, ArrayList<String> commands)
333 334         *
334 335         * @return
335 336         */
336     -     public HashMap<Integer, Server> getServers()
    337 +     public SparseArray<Server> getServers()
337 338         {
338     -         HashMap<Integer, Server> servers = new HashMap<Integer, Server>();
    339 +         SparseArray<Server> servers = new SparseArray<Server>();
339 340
340 341             Cursor cursor = this.getReadableDatabase().query(
341 342                 ServerConstants.TABLE_NAME,
```

Fig. 26 Example of resolving of the `UseSparseArrays` issue (`pocmo/Yaaic` - app/src/main/java/org/yaaic/db/Database.java)

Fig. 27 Example of resolving of the `DrawAllocation` issue (`mchow01/FingerDoodle` - src/edu/cs/tufts/mchow/FingerDoodleView.java)

commit note), the `FloatMath` issue is resolved. In the following we report the commit message provided by the developer when resolving this issue.

"Partially fixed issue with preview on Android 6 in camera2 mode. Fixed nexus naming in CC. Changed deprecated FloatMath to Math." This issue type focuses on the primitives data types, so developers can get rid of this issue by using Math in the source code.

Fig. 28 Example of resolving of the `FloatMath` issue (`almalence/OpenCamera` - src/com/almalence/plugins/capture/panoramaaugmented/PanoramaAugmentedCapturePlugin.java)

Fig. 29 Example of resolving of the `UseValueOf` issue (`AmrutSai/sikuna` - src/uk/org/rivernile/edinburghbustracker/android/MainActivity.java)

UseValueOf Regarding the `UseValueOf` type of issues, following is the commit message developer wrote after resolving this issue from the project *AmrutSai/sikuna*.

"Made changes based on recommendations from the lint tool... and removed unused strings. Preferences have been renamed to Settings. This also no longer appears in the ActionBar on Honeycomb and above... following recommendations at the Android design site. Changed Settings display based on these recommendations also. Made plenty of changes in BusStopDatabase and SettingsDatabase to follow good practices and possible performance enhancements."

As shown in Fig. 29, the developer used a call to `valueOf` (at line 310) to resolve this issue.

References

Ahmed TM, Bezemer C-P, Chen T-H, Hassan AE, Shang W (2016) Studying the effectiveness of application performance management (apm) tools for detecting performance regressions for web applications: an experience report. In: Proceedings of the 13th international conference on mining software repositories. ACM, pp 1–12

Asaduzzaman M, Roy CK, Schneider KA, Di Penta M (2013a) Lhdiff: a language-independent hybrid approach for tracking source code lines. In: 2013 29th IEEE International conference on software maintenance (ICSM). IEEE, pp 230–239

Asaduzzaman M, Roy CK, Schneider KA, Di Penta M (2013b) Lhdiff: tracking source code lines to support software maintenance activities. In: 2013 29th IEEE International conference on software maintenance (ICSM). IEEE, pp 484–487

Ayewah N, Pugh W, Hovemeyer D, Morgenthaler JD, Penix J (2008) Using static analysis to find bugs. IEEE Softw 25(5):22–29

Basili VR, Caldiera G, Rombach HD (1994) The goal question metric approach. In: Encyclopedia of software engineering. Wiley

Bessey A, Block K, Chelf B, Chou A, Fulton B, Hallem S, Henri-Gros C, Kamsky A, McPeak S, Engler D (2010) A few billion lines of code later: using static analysis to find bugs in the real world. Commun ACM 53(2):66–75

Chatzigeorgiou A, Manakos A (2014) Investigating the evolution of code smells in object-oriented systems. ISSE 10(1):3–18

Cohen J (1968) Weighted kappa: nominal scale agreement provision for scaled disagreement or partial credit. Psychol Bull 70(4):213

Couto C, Montandon JE, Silva C, Valente MT (2011) Static correspondence and correlation between field defects and warnings reported by a bug finding tool. Softw Qual J 21:241–257

Cruz L, Abreu R (2017) Performance-based guidelines for energy efficient mobile applications. In: 4th IEEE/ACM International conference on mobile software engineering and systems, MOBILESoft@ICSE 2017, Buenos Aires, Argentina, May 22-23, 2017, pp 46–57

Cruz L, Abreu R (2018) Using automatic refactoring to improve energy efficiency of Android apps, arXiv:1803.05889

Das T, Di Penta M, Malavolta I (2016) A quantitative and qualitative investigation of performance-related commits in Android apps. In: ICSME '16 Proceedings of the 32nd international conference on software maintenance and evolution. IEEE, pp 443–448

Di Penta M, Cerulo L, Aversano L (2009) The life and death of statically detected vulnerabilities: an empirical study. Inform Softw Technol 51(10):1469–1484

Dunn O (1964) Multiple comparisons using rank sums. Technometrics 6:241–252

Foo KC, Jiang ZMJ, Adams B, Hassan AE, Zou Y, Flora P (2015) An industrial case study on the automated detection of performance regressions in heterogeneous environments. In: Proceedings of the 37th international conference on software engineering, vol 2. IEEE Press, pp 159–168

Gómez M, Rouvoy R, Adams B, Seinturier L (2016) Mining test repositories for automatic detection of UI performance regressions in Android apps. In: Proceedings of the 13th international conference on mining software repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016, pp 13–24

Habchi S, Blanc X, Rouvoy R (2018) On adopting linters to deal with performance concerns in android apps. In: ASE18-proceedings of the 33rd IEEE/ACM international conference on automated software engineering, vol 11. ACM Press

Hassan F, Mostafa S, Lam ES, Wang X (2017) Automatic building of java projects in software repositories: a study on feasibility and challenges. In: 2017 ACM/IEEE International symposium on empirical software engineering and measurement (ESEM). IEEE, pp 38–47

Hecht G, Benomar O, Rouvoy R, Moha N, Duchien L (2015a) Tracking the software quality of android applications along their evolution (t). In: 2015 30th IEEE/ACM International conference on automated software engineering (ASE). IEEE, pp 236–247

Hecht G, Rouvoy R, Moha N, Duchien L (2015b) Detecting antipatterns in Android apps. In: 2nd ACM International conference on mobile software engineering and systems, MOBILESoft 2015, Florence, Italy, May 16-17, 2015, pp 148–149

Holm S (1979) A simple sequentially rejective Bonferroni test procedure. Scand J Stat 6:65–70

Huchard M, Kästner C, Fraser G (eds) (2018) Proceedings of the 33rd ACM/IEEE international conference on automated software engineering, ASE 2018, Montpellier, France, September 3-7, 2018. ACM, New York

Joorabchi ME, Mesbah A, Kruchten P (2013) Real challenges in mobile app development. In: 2013 ACM / IEEE International symposium on empirical software engineering and measurement, pp 15–24

Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D (2016) An in-depth study of the promises and perils of mining github. Empir Softw Eng 21(5):2035–2071

Kim S, Ernst MD (2007) Which warnings should I fix first? In: Proceedings of the joint meeting of the European software engineering conference and the ACM SIGSOFT international symposium on foundations of software engineering (ESEC/FSE), pp 45–54

Kruskal WH, Wallis A (1952) Use of ranks in one-criterion variance analysis. J Am Stat Assoc 47:583–621

Leffingwell D (2010) Agile software requirements: lean requirements practices for teams, programs, and the enterprise. Addison-Wesley Professional

Li L, Bissyandé TF, Papadakis M, Rasthofer S, Bartel A, Octeau D, Klein J, Le Traon Y (2017) Static analysis of android apps: a systematic literature review. Information and Software Technology

Lidwell W, Holden K, Butler J (2010) Universal principles of design revised and updated: 125 ways to enhance usability, influence perception, increase appeal, make better design decisions, and teach through design, 2nd edn. Rockport Publishers

Liu Y, Xu C, Cheung S (2014) Characterizing and detecting performance bugs for smartphone applications. In: 36th International conference on software engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014, pp 1013–1024

Malik H, Hemmati H, Hassan AE (2013) Automatic detection of performance deviations in the load testing of large scale systems. In: Proceedings of the 2013 international conference on software engineering. IEEE Press, pp 1012–1021

Munson JC, churn S. G. Elbaum. (1998) Code A measure for estimating the impact of code change. In: International Conference on software maintenance, 1998. Proceedings. IEEE, pp 24–31

Nielson F, Nielson HR, Hankin C (2015) Principles of program analysis. Springer

Nistor A, Ravindranath L (2014) Suncat: helping developers understand and predict performance problems in smartphone applications. In: International symposium on software testing and analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014, pp 282–292

Palomba F, Di Nucci D, Panichella A, Zaidman A, De Lucia A (2017) Lightweight detection of Android-specific code smells: the adoctor project. In: IEEE 24th International conference on software analysis, evolution and reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017, pp 487–491

Palomba F, Vásquez ML, Bavota G, Oliveto R, Di Penta M, Poshyvanyk D, De Lucia A (2018) Crowdsourcing user reviews to support the evolution of mobile apps. J Syst Softw 137:143–162
Pascarella L, Geiger F-X, Palomba F, Di Nucci D, Malavolta I, Bacchelli A (2018) Self-reported activities of android developers. In: 5th IEEE/ACM International conference on mobile software engineering and systems, page to appear. ACM, New York
Qian F, Wang Z, Gerber A, Mao ZM, Sen S, Spatscheck O (2011) Profiling resource usage for mobile applications: a cross-layer approach. In: Proceedings of the 9th international conference on mobile systems, applications, and services (MobiSys 2011), Bethesda, MD, USA, June 28 - July 01, 2011, pp 321–334
Ravindranath L, Padhye J, Agarwal S, Mahajan R, Obermiller I, Shayandeh S (2012) Appinsight: mobile app performance monitoring in the wild. In: 10th USENIX Symposium on operating systems design and implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012, pp 107–120
Reimann J, Brylski M, Aßmann U (2014) A tool-supported quality smell catalogue for Android developers. Softwaretechnik-Trends, 34(2)
Rosenberg J (2008) Statistical methods and measurement. In: Guide to advanced empirical software engineering. Springer, pp 155–184
Sadowski C, Aftandilian E, Eagle A, Miller-Cushon L, Jaspan C (2018) Lessons from building static analysis tools at google. Commun ACM 61(4):58–66
Shull F, Singer J, Sjøberg DI (2007) Guide to advanced empirical software engineering. Springer
Spacco J, Hovemeyer D, Pugh W (2006) Tracking defect warnings across versions. In: Proceedings of the 2006 international workshop on mining software repositories. ACM, pp 133–136
Spencer D (2009) Card sorting: designing usable categories. Rosenfeld Media
Sulír M, Porubän J (2016) A quantitative study of java software buildability. In: Proceedings of the 7th international workshop on evaluation and usability of programming languages and tools. ACM, pp 17–25
Tufano M, Palomba F, Bavota G, Oliveto R, Di Penta M, De Lucia A, Poshyvanyk D (2017) When and why your code starts to smell bad (and whether the smells go away). IEEE Trans Softw Eng 43(11):1063–1088
Tufano M, Watson C, Bavota G, Di Penta M, White M, Poshyvanyk D (2019) An empirical study on learning bug-fixing patches in the wild via neural machine translation. ACM Trans Softw Eng Methodol 28(4):19,1–19,29. https://doi.org/10.1145/3340544
Vásquez ML, Vendome C, Luo Q, Poshyvanyk D (2015) How developers detect and fix performance bottlenecks in Android apps. In: 2015 IEEE International conference on software maintenance and evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015, pp 352–361
Wedyan F, Alrmuny D, Bieman JM (2009) The effectiveness of automated static analysis tools for fault detection and refactoring prediction. In: 2009 International conference on software testing verification and validation. IEEE, pp 141–150
Wohlin C, Runeson P, Höst M, Ohlsson M, Regnell B, Wesslén A (2012) Experimentation in software engineering. Computer Science. Springer
Zaman S, Adams B, Hassan AE (2012) A qualitative study on performance bugs. In: Proceedings of the 9th IEEE working conference on mining software repositories. IEEE Press, pp 199–208

Teerath Das is PhD student in Computer Science at the Gran Sasso Science Institute (GSSI), L'Aquila, Italy. Prior to this, he received his M.Sc. in Computer Science from South Asian University, India and B.E in Computer Engineering from NUST Rawalpindi in 2014 and 2012, respectively. His area of research include: Empirical Software Engineering, Mining Software Repositories (mainly from GitHub), and Software Evolution and Maintenance. He is applying these research techniques to empirically analyse performance-related issues in open-source Android applications. He coauthorized several scientific peer-reviewed articles in international conferences and journals.

Massimiliano Di Penta is full professor at the University of Sannio, Italy. His research interests include software maintenance and evolution, mining software repositories, empirical software engineering, search-based software engineering, and software testing. He is an author of over 250 papers appeared in international journals, conferences, and workshops. He serves and has served in the organizing and program committees of more than 100 conferences, including ICSE, FSE, ASE, ICSME. He is co-editor in Chief of the Journal of Software: Evolution and Processes edited by Wiley, editorial board member of ACM Transactions on Software Engineering and Methodology and Empirical Software Engineering Journal edited by Springer, and has served the editorial board of the IEEE Transactions on Software Engineering.

Ivano Malavolta is assistant professor at the Computer Science Department of the Vrije Universiteit Amsterdam (The Netherlands). His research focuses on data-driven software engineering, with a special emphasis on software architecture, mobile software development, robotics software. He applies empirical methods to assess practices and trends in the field of software engineering. He authored several scientific articles in international journals and peerreviewed international conferences proceedings. He is program committee member and reviewer of international conferences and journals in the software engineering field. He received a PhD in computer science from the University of L'Aquila in 2012. He is a member of ACM, IEEE, VERSEN, Amsterdam Young Academy, and Amsterdam Data Science.